RC 22208 (98723) February 13, 2001 **Computer Science**

IBM Research Report

Multidimensional Indexing Structures for Content-based Retrieval

Vittorio Castelli
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Multidimensional Indexing Structures for Content-Based Retrieval

Vittorio Castelli

October 4, 2001

1 Introduction

Indexing plays a fundamental role in supporting efficient retrieval of sequences of images, of individual images and of selected subimages from multimedia repositories.

Three categories of information are extracted and indexed in image databases: metadata, objects and features, and relations between objects [129]. This chapter is devoted to indexing structures for objects and features.

Content-based retrieval (CBR) of imagery has become synonym of retrieval based on low-level descriptors, such as texture, color and shape. Similar images map to high-dimensional feature vectors that are close to each other in terms of Euclidean distance. A large body of literature exists on the topic, and different aspects have been extensively studied, including the selection of appropriate metrics, the inclusion of the user in the retrieval process, and, particularly, on indexing structures to support query-by-similarity.

Indexing of metadata and relations between objects are not covered here, since their scope far exceeds image databases. Metadata indexing is a complex application-dependent problem. Active research areas include the automatic extraction of information from unstructured textual description, the definition of standards (for example, for remotely sensed images), and the translation between different standards (such as in medicine). The techniques required to store and retrieve spatial relations from images are analogous to those used in geographic information systems (GIS), and the topic has been extensively studied in this context.

This chapter is organized as follows. The current section is concluded by a paragraph on notation. Section 2 is devoted to background information on representing images using low-level features. Section 3 introduces three taxonomies of indexing methods, two of which are used to provide primary and secondary structure to the following Section 4.1, dealing with

vector-space methods, and Section 4.2, which describe metric-space approaches. Section 5 contains a discussion on how to select among different indexing structures. Conclusions and future directions are in Section 6. The Appendix contains a description of numerous methods introduced in Section 4.

The bibliography that concludes the chapter also contains numerous references not directly cited in the text.

1.1 Notation

A database or a database table $\boldsymbol{\mathcal{X}}$ is a collection of n items that can be represented in a $d-$dimensional real space, denoted by $\mathbb{R}^d$. Individual items that have a spatial extent are often approximated by a minimum bounding rectangle (MBR) or by some other representation. The other items, such as vectors of features, are represented as points in the space. Points in a $d-$dimensional space are in $1:1$ correspondence with vectors centered at the origin, and therefore the words vector, point, and database item are used interchangeably. A vector is denoted by a lower-case bold face letter, as in $\mathbf{x}$, and the individual components are identified using the square bracket notation; thus $\mathbf{x}[i]$ is the ith component of the vector $\mathbf{x}$. Upper case bold letters are used to identify matrices; for instance, $\mathbf{I}$ is the identity matrix. Sets are denoted by curly brackets enclosing their content, like in $\{A, B, C\}$. The desired number of nearest neighbors in a query is always denoted by k. The maximum depth of a tree is denoted by L, while the dummy variable for level is ℓ.

A significant body of research is devoted to retrieval of images based on low-level features (such as shape, color and texture,) represented by descriptors – numerical quantities computed from the image that try to capture specific visual characteristics. For example, the color histogram and the color moments are descriptors of the color feature. Since in the literature the terms "feature" and "descriptor" are almost invariably used as synonyms, we will also use them interchangeably.

2 Feature-Level Image Representation

In this section, we discuss several different aspects of feature-level image representation. First we contrast full image match and subimage match, and discuss the corresponding feature extraction methodologies. We then describe a taxonomy of query types used in content-based retrieval systems. We next discuss the concept of distance function as a means of computing similarity between images, represented as high-dimensional vectors of features. When dealing with high-dimensional spaces our geometric intuition is extremely misleading. The nice properties of the low-dimensional spaces we are familiar with do not carry over to high-dimensional spaces, and a class of phenomena arises, known as the "curse of dimensionality", to which we devote a section. A way of coping with the curse of dimensionality is to reduce

the dimensionality of the search space, and appropriate techniques are discussed in Section 2.5.

2.1 Full Match, Subimage Match and Image Segmentation

Similarity retrieval can be divided into *whole image match*, where the query template is an entire image and is matched against entire images in the repository, and *subimage match*, where the query template is a portion of an image and the results are portions of images from the database. A particular case of subimage match consists of retrieving portions of images containing desired objects.

Whole match is the most commonly used approach to retrieve photographic images. A single vector of features, which are represented as numeric quantities, is extracted from each image, and used for indexing purposes. Early content-based retrieval systems such as QBIC [126], adopt this framework.

Subimage match is more important in scientific datasets, such as remotely sensed images, medical images, or seismic data for the oil industry, where the individual images are extremely large (several hundred megabytes or larger) and the user is generally interested in subsets of the data (e.g., regions showing beach erosion, portions of the body surrounding a particular lesion etc.).

Most existing systems support subimage retrieval by segmenting the images at database ingestion time, and associating a feature vector with each interesting portion. Segmentation can be data-independent (windowed or block-based), or data-dependent (adaptive).

Data-independent segmentation commonly consists of dividing an image into overlapping or non-overlapping, fixed-size sliding rectangular regions of equal stride, and extracting and indexing a feature vector from each such region [107, 35]. The selection of the window size and stride are application-dependent. For example, in [107] texture features are extracted from satellite images using non-overlapping square windows of size 32×32, while in [61] texture is extracted from well-bore images acquired with the Formation Micro-scanner Imager, which are 192 pixel wide and tens-to-hundreds of thousand pixels high. Here the extraction windows have size 24×32, have a horizontal stride of 24, and a vertical stride of 2.

Numerous approaches to data-dependent feature extraction have been proposed. The *blobworld* representation [37] (where images are segmented using simultaneously color and texture features by an Expectation-Maximization (EM) algorithm [57],) is well-tailored towards identifying objects in photographic images, provided that they stand out from the background. Each object is efficiently represented by replacing it with a "blob" - an ellipse identified by its centroid and its scatter matrix. The mean texture and the two dominant colors are extracted and associated with each blob. The *Edge-Flow* algorithm [116, 115] is designed to produce an exact segmentation of an image by using a smoothed texture field and predictive coding to identify points where edges exist with high probability. The *MMAP* algorithm [154] divides the image into overlapping rectangular regions, extracts from each

regions a feature vector, quantizes is, constructs a cluster index map by representing each window with the label produced by the quantizer, and applies a simple random field model to smooth the cluster index map. Connected regions having the same cluster label are then indexed by the label.

Adaptive feature extraction produces a much smaller feature volume than data-independent block-based extraction, and the ensuing segmentation can be used for automatic semantic labeling of image components. It is typically less flexible than image-independent extraction, since images are partitioned at ingestion time. Block-based feature extraction yields a larger number of feature vectors per image, and can allow very flexible, query-dependent segmentation of the data (this is not surprising, since often a block-based algorithm is the first step of an adaptive one). An example is presented in [22, 61] where the system retrieves subimages containing objects which are defined by the user at query specification time and constructed during the execution of the query using finely-gridded feature data.

2.2 Types of Content-Based Queries

In this section, we describe the different types of queries typically used for content-based search.

The search methods used for image databases differ from those of traditional databases. Exact queries are only of moderate interest, and, when they apply, are usually based on metadata, managed by a traditional Data Base Management System (DBMS). The quintessential query method for multimedia databases is *retrieval-by-similarity*. The user search, expressed through one of a number of possible user interfaces, is translated into a query on the feature table or tables. We group similarity queries into three main classes:

1. **Range search:** "Find all images where feature 1 is within range r_1, and feature 2 is within range r_2, and ..., and feature n is within range r_n." Example: *Find all images showing a tumor of size between* $size_{\min}$ *and* $size_{\max}$*, within a given region.*

2. $k-$**Nearest-neighbor search:** "Find the k most similar images to the template". Example: *Find the* 20 *tumors that are most similar to a specified example, where similarity is defined in terms of location, shape and size, and return the corresponding images.*

3. **Within-distance (or** α**-cut):** "Find all images with a similarity score better than α with respect to a template", or "Find all images at distance less than d from a template". Example: *Find all the images containing tumors with similarity scores larger than* α_0 *with respect to an example provided.*

This categorization is the fundamental taxonomy used in this chapter.

Note that nearest-neighbor queries are required to return at least k results, possibly more in case of ties, no matter how similar the results are to the query, while within-distance queries do not have an upper bound on the number of returned results, but are allowed to return an empty set. A query of type 1 requires a complex interface or a complex query language, such as SQL. Queries of type 2 and 3 can, in their simplest incarnations, be expressed through the use of simple, intuitive interfaces that support query-by-example.

Nearest-neighbor queries (type 2) rely on the definition of a *similarity function*. Section 2.3 is devoted to the use of distance functions for measuring similarity. Nearest-neighbor search problems have wide applicability beyond information retrieval and GIS data management. There is a vast literature dealing with nearest-neighbor problems in the fields of pattern recognition, supervised learning, machine learning, and statistical classification [54, 59, 58, 55], as well as in the areas of unsupervised learning, clustering and vector quantization [109, 140, 141].

$\alpha-$cut queries (type 3) rely on a distance or *scoring function*. A scoring function is non-negative, bounded from above, and assigns higher values to better matches. For example, a scoring function might order the database records by how well they match the query, and then use the record rank as the score. The last record, which is the one that best satisfies the query, has the highest score. Scoring functions are commonly normalized between 0 and 1.

In our discussion, we have implicitly assumed that query processing has three properties[1]:

Exhaustiveness – Query processing is exhaustive if it retrieves all the database items satisfying it. A database item that satisfies the query and does not belong to the result set is called a *miss*. Non-exhaustive range-query processing fails to return points that lie within the query range. Non-exhaustive $\alpha-$cut-query processing fails to return points that are closer than α to the query template. Non-exhaustive k-nearest-neighbor-query processing either returns fewer than k results, or returns results that are not correct.

Correctness – Query processing is correct if all the returned items satisfy it. A database item that belongs to the result set and does not satisfy the query is called a *false hit*. Non-correct range-query processing returns points outside the specified range. Non-correct $\alpha-$cut-query processing returns points that are farther than α from the template. Non-correct $k-$nearest-neighbor query processing miss some of the desired results, and therefore is also non-exhaustive.

[1]In this chapter we restrict the attention to properties of indexing structures. The content-based retrieval community has concentrated mostly on properties of the image-representation: as discussed in other chapters, numerous studies have investigated how well different feature-descriptor sets perform by comparing results selected by human subjects with results retrieved using features. Different feature sets produce different number of misses and of false hits, and have different effects on the result rankings. In this chapter we are not concerned with the performance of feature descriptors: an indexing structure that is guaranteed to return exactly the k nearest feature vectors of every query, is, for the purpose of this chapter, exhaustive, correct, and deterministic. This same indexing structure, used in conjunction with a specific feature set might yield query results that a human would judge as misses, false hits, or incorrectly ranked.

Determinism – Query processing is deterministic if it returns the same results every time a query is issued, and for every construction of the index[2]. It is possible to have non-deterministic range, $\alpha-$cut and k-nearest-neighbor queries.

We will use the term *exactness* to denote the combination of exhaustiveness and correctness. It is very difficult to construct indexing structures that have all three properties and that are at the same time efficient (namely, that perform better than brute-force sequential scan) as the dimensionality of the dataset grows. Much can be gained, however, if one or more of the assumptions are relaxed.

Relaxing Exhaustiveness Relaxing exhaustiveness alone means allowing misses but not false hits, and retaining determinism. There is a widely used class of non-exhaustive methods that do not modify the other properties. These methods support **fixed-radius queries**, namely, they return only results that have distance smaller than r from the query point. The radius r is either fixed at index construction time, or specified at query time. Fixed-radius k-nearest-neighbor queries are allowed to return less than k results, if less than k database points lie within distance r of the query sample.

Relaxing Exactness It is impossible to give up correctness in nearest-neighbor queries while retaining exhaustiveness, and we are not aware of methods that achieve this goal for $\alpha-$cut and range queries. There are two main approaches to relax exactness:

- **$1 + \epsilon$ queries** return results whose distance is guaranteed to be less than $1 + \epsilon$ times the distance of the exact result;
- **approximate queries** operate on an approximation of the search space, obtained, for instance, through dimensionality reduction (Section 2.5).

Approximate queries usually constrain the average error, while $1 + \epsilon$ queries limit the maximum error. Note that it is possible to combine the approaches, for instance by first reducing the dimensionality of the search space, and indexing the result with a method supporting $1 + \epsilon$ queries.

Relaxing Determinism There are three main categories algorithms yielding non-deterministic indexes where the lack of determinism is due to a randomization step in the index construction [50, 100]:

- methods yielding indexes that relax exhaustiveness or correctness, and that are slightly different every time the index is constructed – repeatedly reindexing the same database produces indexes with very similar but not identical retrieval characteristics;

[2]While this definition may appear cryptic, it will soon be clear that numerous approaches exist that yield non-deterministic queries.

- methods yielding yields "good" indexes (e.g., both exhaustive and correct) with arbitrarily high probability, and poor indexes with low probability – repeatedly reindexing the same database yields mostly indexes with the desired characteristics, and very rarely an index that performs poorly;
- methods whose indexes perform well (e.g., are both exhaustive and correct) on the vast majority of queries, and poorly on the remaining – if queries are generated "at random", with high probability the results will be accurate.

A few non-deterministic methods rely on a randomization step during the query *execution* – the same query on the same index might not return the same results.

Exhaustiveness, exactness and determinism can be individually relaxed for all three main categories of queries. It is also possible to relax any combination of these properties: for example CSVD (described in Appendix A) supports nearest-neighbor searches that are both non-deterministic and approximate.

2.3 Image Representation and Similarity Measures

In general, systems supporting $k-$nearest-neighbor and $\alpha-$cut queries rely on the following assumption:

"*Images (or image portions) can be represented as points in an appropriate metric space where dissimilar images are distant from each other, similar images are close to each other, and where the distance function captures well the user's concept of similarity.*"

Since query-by-example has been the main approach to content-based search, a substantial literature exists on how to support nearest-neighbor and $\alpha-$cut searches, both of which rely on the concept of distance (a score is usually directly derived from a distance). A *distance function* (or *metric*) $D(\cdot,\cdot)$ is by definition non-negative, symmetric, satisfies the triangular inequality, and has the property that $D(x,y)=0$ if and only if $x=y$. A metric space is a pair of items: a set $\boldsymbol{\mathcal{X}}$, the elements of which are called points, and a distance function defined on pairs of elements of $\boldsymbol{\mathcal{X}}$.

The problem of finding a universal metric that acceptably captures photographic image similarity as perceived by human beings is unsolved and indeed ill-posed, since subjectivity plays a major role in determining similarities and dissimilarities. In specific areas, however, objective definitions of similarity can be provided by experts, and in these cases it might be possible to find specific metrics that solve exactly the problem.

When images or portions of images are represented using a collection of d features $\mathbf{x}[1],\ldots,\mathbf{x}[d]$ (containing texture, shape, color descriptors or combinations thereof), it seems natural to aggregate the features into a vector (or, equivalently, a point) in the $d-$dimensional space $\mathbb{R}^d$, by making each feature correspond to a different coordinate axis. Some specific

features, such as the color histogram, can be interpreted both as point and as probability distributions.

Within the vector representation of the query space, executing a range query is equivalent to retrieving all the points lying within a hyperrectangle aligned with the coordinate axes. To support nearest-neighbor and $\alpha-$cut queries, however, the space must be equipped with a metric or a dissimilarity measure. Note that, although the dissimilarity between statistical distributions can be measured with the same metrics used for vectors, there are also dissimilarity measures that were specifically developed for distributions.

We now describe the most common dissimilarity measures, provide their mathematical form, discuss their computational complexity, and mention when they are specific to probability distributions.

Euclidean or $D^{(2)}$. Computationally simple (O(d) operations) and invariant with respect to rotations of the reference system, the Euclidean distance is defined as

$$D^{(2)}(\mathbf{x},\mathbf{y}) = \sqrt{\sum_{i=1}^{d} (\mathbf{x}[i]-\mathbf{y}[i])^2}.$$

Rotational invariance is important in dimensionality reduction, as discussed in Section 2.5. The Euclidean distance is the only rotationally invariant metric in this list (the rotationally invariant correlation coefficient described later is not a distance). The set of vectors of length d having real entries, endowed with the Euclidean metric is called the $d-$dimensional Euclidean space. When d is a small number, the most expensive operation is the square root. Hence, the square of the Euclidean distance is also commonly used to measure similarity.

Chebychev or $D^{(\infty)}$. Less computationally expensive than the Euclidean distance (but still requiring O(d) operations), it is defined as

$$D^{(\infty)}(\mathbf{x},\mathbf{y}) = \max_{i=1}^{d} |\mathbf{x}[i]-\mathbf{y}[i]|.$$

Manhattan or $D^{(1)}$ or city-block. As computationally expensive as a squared Euclidean distance, this distance is defined as

$$D^{(1)}(\mathbf{x},\mathbf{y}) = \sum_{i=1}^{d} |\mathbf{x}[i]-\mathbf{y}[i]|.$$

Minkowsky or $D^{(p)}$. This is really a family of distance functions, parameterized by p. The three previous distances belong to this family, and correspond to $p=2$, $p=\infty$ (interpreted as $\lim_{p\to\infty} D^p$) and $p=1$ respectively.

$$D^{(p)}(\mathbf{x},\mathbf{y}) = \left[\sum_{i=1}^{d} |\mathbf{x}[i]-\mathbf{y}[i]|^p\right]^{\frac{1}{p}}.$$

Minkowsky distances have the same number of additions and subtractions as the Euclidean distance. With the exception of D^1, D^2 and D^∞, the main computational cost is due to computing the power functions. Often Minkowsky distances between functions are also called L_p distances, and Minkowsky distances between finite or infinite sequences of numbers are called l_p distances.

Weighted Minkowsky. Again, this is a family of distance functions, parameterized by p, where the individual dimensions can be weighted differently using non-negative weights w_i. Their mathematical form is

$$D_{\mathbf{w}}^{(p)}(\mathbf{x}, \mathbf{y}) = \left[\sum_{i=1}^{d} w_i \left| \mathbf{x}[i] - \mathbf{y}[i] \right|^p \right]^{\frac{1}{p}}.$$

The Weighted Minkowsky distances require d more multiplications than their unweighted counterpart.

Mahalanobis. A computationally expensive generalization of the Euclidean distance, it is defined in terms of a covariance matrix $\mathbf{C}$

$$D(\mathbf{x}, \mathbf{y}) = |\det \mathbf{C}|^{1/d} (\mathbf{x} - \mathbf{y})^T \mathbf{C}^{-1} (\mathbf{x} - \mathbf{y}),$$

where det is the determinant, $\mathbf{C}^{-1}$ is the matrix inverse of $\mathbf{C}$, and the superscript T denotes transposed. If $\mathbf{C}$ is the identity matrix $\mathbf{I}$, the Mahalanobis distance reduces to the Euclidean distance squared, otherwise the entry $\mathbf{C}[i, j]$ can be interpreted as the joint contribution of the ith and jth feature to the overall dissimilarity. In general the Mahalanobis distance requires $O(d^2)$ operations. This metric is also commonly used to measure the distance between probability distributions.

Generalized Euclidean or quadratic, is a generalization of the Mahalanobis distance where the matrix $\mathbf{K}$ is positive definite but not necessarily a covariance matrix, and the multiplicative factor is omitted:

$$D(\mathbf{x}, \mathbf{y}) = (\mathbf{x} - \mathbf{y})^T \mathbf{K} (\mathbf{x} - \mathbf{y}).$$

It requires $O(d^2)$ operations.

Correlation Coefficient. Defined as

$$\rho(\mathbf{x}, \mathbf{y}) = \frac{\sum_{i=1}^{d} (\mathbf{x}[i] - \overline{\mathbf{x}}[i]) (\mathbf{y}[i] - \overline{\mathbf{x}}[i])}{\sqrt{\sum_{i=1}^{d} (\mathbf{x}[i] - \overline{\mathbf{x}}[i])^2 \sum_{i=1}^{d} (\mathbf{y}[i] - \overline{\mathbf{x}}[i])^2}},$$

(where $\overline{\mathbf{x}} = [\overline{\mathbf{x}}[1], \ldots, \overline{\mathbf{x}}[d]]$ is the average of all the vectors in the database,) the correlation coefficient is not a distance. However, if the points $\mathbf{x}$ and $\mathbf{y}$ are projected onto the sphere of unit radius centered at $\overline{\mathbf{x}}$, then the quantity $2 - 2\rho(\mathbf{x}, \mathbf{y})$ is exactly the Euclidean distance between the projections. The correlation coefficient is invariant with respect to rotations and scaling of the search space. It requires $O(d)$ operations. This measure of similarity is used in statistic to characterize the joint behavior of pairs of random variables.

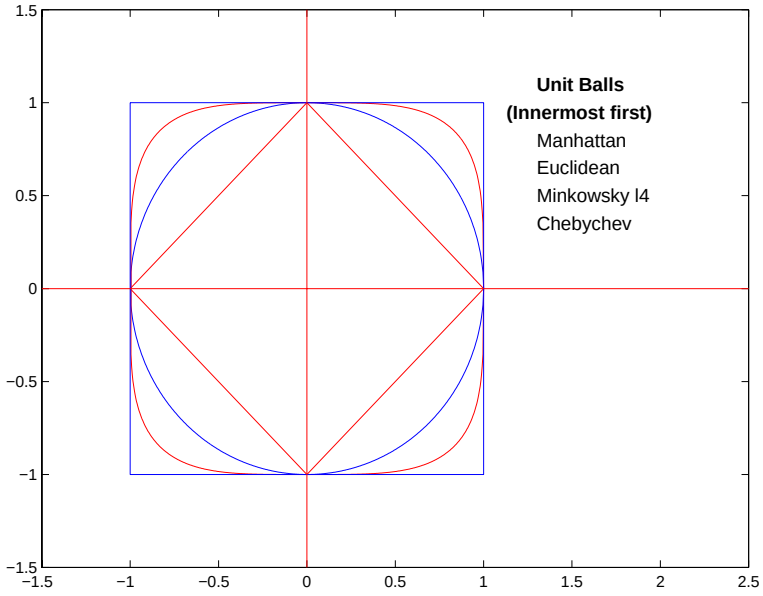

Figure 1: The unit spheres under Chebychev, Euclidean, $D^{(4)}$ and Manhattan distance

Relative Entropy or Kullback-Leibler Divergence. This information-theoretical quantity is defined, only for probability distributions, as

$$D(\mathbf{x}\|\mathbf{y}) = \sum_{i=1}^{d} \mathbf{x}[i] \log \frac{\mathbf{x}[i]}{\mathbf{y}[i]}.$$

It is meaningful only if the entries of $\mathbf{x}$ and $\mathbf{y}$ are non-negative and $\sum_{i=1}^{d} \mathbf{x}[i] = \sum_{i=1}^{d} \mathbf{y}[i] = 1$. Its computational cost is $O(d)$, however it requires $O(d)$ divisions and $O(d)$ logarithm computations. It is not a distance, since it is not symmetric, nor it satisfies a triangle inequality. When used for retrieval purposes, the first argument should be the query vector, and the second argument the database vector. It is also known as Kullback-Leibler distance, Kullback-Leibler cross-entropy or just as cross-entropy.

χ^2−Distance. Defined, only for probability distributions, as

$$D_{\chi^2}(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{d} \frac{\mathbf{x}^2[i] - \mathbf{y}^2[i]}{\mathbf{y}[i]}.$$

It lends itself to a natural interpretation only if the entries of $\mathbf{x}$ and $\mathbf{y}$ are non-negative and $\sum_{i=1}^{d} \mathbf{x}[i] = \sum_{i=1}^{d} \mathbf{y}[i] = 1$. Computationally, it requires $O(d)$ operations, the most expensive of which is the division. It is not a distance, since it is not symmetric.

It is difficult to convey an intuitive notion of the difference between distances. Concepts derived from geometry can assist in this task. As in topology, where the structure of a

topological space is completely determined by its open sets, the structure of a metric space is completely determined by its balls. A ball centered at $\mathbf{x}$ having radius r is the set of points having distance r from $\mathbf{x}$. The Euclidean distance is the starting point of our discussion, since it can be measured using a ruler. Balls in Euclidean spaces are the spherical surfaces we are familiar with (Figure 1). A ball in D^∞ is a hyper-square aligned with the coordinate axes, inscribing the corresponding Euclidean ball. A ball in D^1 is a hyper-square having vertices on the coordinate axes and inscribed in the corresponding Euclidean ball. A ball in D^p, for $p > 2$ looks like a "fat sphere" that lies between the D^2 and D^∞ balls, while for $1 < p < 2$ lies between the D^1 and D^2 balls, and looks like a "slender sphere". It is immediately possible to draw several conclusions. Consider the distance between two points $\mathbf{x}$ and $\mathbf{y}$, and look at the absolute values of the differences $d_i = |\mathbf{x}[i] - \mathbf{y}[i]|$.

- The Minkowsky distances differ in the way they combine the contributions of the d_i's. All the d_i's contribute equally to $D^1(\mathbf{x}, \mathbf{y})$, irrespective of their values. However, as p grows, the value $D^p(\mathbf{x}, \mathbf{y})$ is increasingly determined by the maximum of the d_i, while the overall contribution of all the other differences becomes less and less relevant. In the limit, $D^\infty(\mathbf{x}, \mathbf{y})$ is uniquely determined by the maximum of the differences d_i, while all the other values are ignored.

- If two points have distance D^p equal to zero, for some $p \in [1\infty]$, then they have distance D^q equal to zero for all $q \in [1, \infty]$. Hence, one cannot distinguish points that have, say, Euclidean distance equal to zero by selecting a different Minkowsky metric.

- If $1 \leq p < q \leq \infty$ the ratio $D^p(\mathbf{x}, \mathbf{y})/D^q(\mathbf{x}, \mathbf{y})$, is bounded from above $K_{p,q}$ and from below by 1. The constant $K_{p,q}$ is never larger than 2^d and depends only on p and q, but not on $\mathbf{x}$ and $\mathbf{y}$. This property is called *equivalence* of distances. Hence, there are limits on how much the metric structure of the space can be modified by the choice of Minkowsky distance.

- Minkowsky distances do not take into account combinations of d_i's. In particular, if two features are highly correlated, differences between the values of the first feature are likely reflected in distances between the values of the second feature. The Minkowsky distance combines the contribution of both differences, and can overestimate visual dissimilarities.

We argue that Minkowsky distances are substantially similar to each other from the viewpoint of information retrieval, and that there are very few theoretical arguments supporting the selection of one over the others. Computational cost and rotational invariance are probably more important considerations in the selection.

If the covariance matrix $\mathbf{C}$ and the matrix $\mathbf{K}$ have full rank, and the weights w_i are all positive, then the Mahalanobis distance, the Generalized Euclidean distance and the unweighted and weighted Minkowsky distances are equivalent.

Weighted $D^{(p)}$ distances are useful when different features have different ranges. For instance, if a vector of features contains both the fractal dimension (which takes values between 2 and 3) and the variance of the gray-scale histogram (which takes values between 0 and 2^{14} for an 8-bit image), the latter will be by far the main factor in determining the $D^{(p)}$ distance between different images. This problem is commonly corrected by selecting an appropriate weighted $D^{(p)}$ distance. Often each weight is the reciprocal of the standard deviation of the corresponding feature computed across the entire database.

The Mahalanobis distance solves a different problem. If two features i and j have significant correlation, then $|\mathbf{x}[i] - \mathbf{y}[i]|$ and $|\mathbf{x}[j] - \mathbf{y}[j]|$ are correlated: if $\mathbf{x}$ and $\mathbf{y}$ differ significantly in the ith dimension, they will likely differ significantly in the jth dimension, and if they are similar in one dimension, they will likely be similar in the other dimension. This means that the two features capture very similar characteristics of the image. When both features are used in a regular or weighted Euclidean distance, the same dissimilarities are essentially counted twice. The Mahalanobis distance offers a solution consisting of correcting for correlations and differences in dispersion around the mean. A common use of this distance is in classification applications, where the distributions of the classes are assumed to be Gaussian. Both Mahalanobis distance and Generalized Euclidean distances have unit spheres shaped as ellipsoids, aligned with the eigenvectors of the weights matrices.

The characteristics of the problem being solved should suggest the selection of a distance metric. In general, the Minkowsky distance considers only the dimension where $\mathbf{x}$ and $\mathbf{y}$ differ the most, the Euclidean distance captures our geometric notion of distance, and the Manhattan distance combines the contributions of all dimensions where $\mathbf{x}$ and $\mathbf{y}$ are different. Mahalanobis distances and Generalized Euclidean distances consider joint contributions of different features.

Empirical approaches exist, typically consisting of constructing a set of queries, for which the correct answer is determined manually, and comparing different distances in terms of efficiency and accuracy. Efficiency and accuracy are often measured using the information-retrieval quantities *precision* and *recall* defined as follows. Let $\mathbb{D}$ be the set of desired (correct) results of a query, usually manually selected by a user, and $\mathbb{A}$ be the set of actual query results. We require that $|\mathbb{A}|$ be larger than $|\mathbb{D}|$. Some of the results in $\mathbb{A}$ will be correct, and form a set $\mathbb{C}$. Precision and recall for individual queries are then respectively defined as

$$P = \frac{|\mathbb{C}|}{|\mathbb{A}|} \quad = \text{fraction of returned results that are correct;}$$

$$R = \frac{|\mathbb{C}|}{|\mathbb{D}|} \quad = \text{fraction of correct results that are returned.}$$

Precision and recall of individual queries are For the approach to be reliable, the database size should be very large, and precision and recall should be averaged over a large number of queries.

J.R. Smith [153] observed that, on a medium-sized, diverse photographic image database, and for a heterogeneous set of queries, when retrieval is based on color histogram or

on texture, precision and recall vary only slightly with the choice of (Minkowsky or weighted Minkowsky) metric.

2.4 The "Curse of Dimensionality"

The operations required to perform content-based search are computationally expensive, and therefore indexing schemes are commonly used to speed up the queries.

Indexing multimedia databases is a much more complex and difficult problem than indexing traditional databases. The main difficulty stems from using long feature vectors to represent the data. This is especially troublesome in systems supporting only whole image matches where individual images are represented using extremely long feature vectors.

Our geometric intuition (based on experience with the 3-dimensional world in which we live) leads us to believe that numerous geometric properties hold in high-dimensional spaces, while in reality they cease to be true very early on as the number of dimensions grows. For example, in 2 dimensions a circle is well approximated by the minimum bounding square: the ratio of the areas is $4/\pi$. However, in 100 dimensions the ratio of the volumes becomes approximately $4.2 \cdot 10^{39}$: most of the volume of a 100-dimensional hypercube is outside the largest inscribed sphere – hypercubes are poor approximations of hyperspheres, and the large majority of indexing structures partition the space into hypercubes or hyperrectangles.

Two classes of problems then arise. The first is algorithmic: indexing schemes that rely on properties of low-dimensionality spaces do not perform well in high-dimensional spaces because the assumptions on which they are based do not hold there. For example, R-trees are extremely inefficient for performing $\alpha-$cut queries using the Euclidean distance, because they execute the search by transforming it into the range query defined by the minimum bounding rectangle of the desired search region, which is a sphere centered on the template point, and by checking whether the retrieved results satisfy the query. In high dimensions, the R-tree retrieve mostly irrelevant points, that lie within the hyperrectangle but outside the hypersphere.

The second class of difficulties, called the "curse of dimensionality", is intrinsic in the geometry of high-dimensional hyperspaces, which entirely lack the"nice" properties of low-dimensional spaces.

One of the characteristics of high-dimensional spaces is that points randomly sampled from the same distribution appear uniformly far from each other, and each point sees itself as an outlier (see [46, 24, 26, 92, 73] for formal discussions of the problem). More specifically, a randomly selected database point does not perceive itself as surrounded by the other database points; on the contrary, the vast majority of the other database vector appear to be almost at the same distance and to be located in the direction of the center. Note that, while the semantics of range queries are unaffected by the curse of dimensionality, the meaning itself of nearest-neighbor and of $\alpha-$cut queries is now in question.

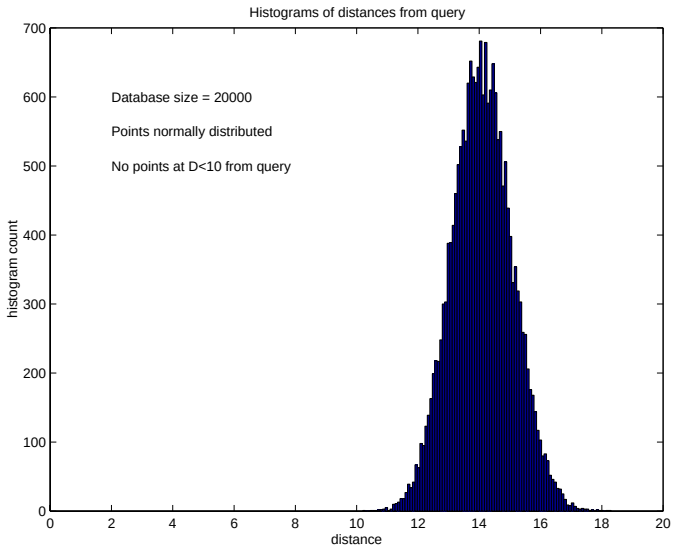

Figure 2: Distances between a query point and database points. Database size = 20,000 points, in 100 dimensions.

Consider the following simple example: let a database be composed of 20,000 independent 100-dimensional vectors, with the features of each vector independently distributed as standard Normal random (i.e., Gaussian) variables. Normal distributions are very concentrated: the tails decay extremely fast, and the probability of sampling observations far from the mean is negligible. A large Gaussian sample in 3-dimensional space looks like a tight, well concentrated cloud, a nice "cluster". Not so in 100 dimensions. In fact, sampling an independent query template according to the same 100-dimensional standard Normal, and computing the histogram of the distances between this query point and the points in the database, yields the result shown in Figure 2. In the data used for the figure, the minimum distance between the query and a database point is 10.1997, and the maximum distance is 18.3019. There are no "close" points to the query, and there are no "far" points from the query. A $\alpha-$cut queries become very sensitive to the choice of the threshold. With a threshold smaller than 10, no result is returned; with a threshold of 12.5, the query returns 5.3% of the database; barely increasing the threshold to to 13 returns almost three times as many results, 14% of the database.

2.5 Dimensionality Reduction

If the high-dimensional representation of images actually behaved as described in the previous section, queries of type 2 and 3 would be essentially meaningless. Luckily, two properties come to the rescue. The first, noted in [24] and, from a different perspective, in [157, 156], is that the feature space often has a local structure, thanks to which query images have, in fact, close neighbors, and therefore nearest-neighbor and $\alpha-$cut searches can be meaningful. The second is that the features used to represent the images are usually not independent, and are

often highly correlated: the feature vectors in the database can be well approximated by their "projections" onto a lower-dimensionality space, where classical indexing schemes work well. Pagel, Korn and Faloutsos [137] propose a method for measuring the intrinsic dimensionality of data sets in terms of their fractal dimensions. By observing that the distribution of real data often displays self-similarity at different scales, they express the average distance of the kth nearest neighbor of a query sample in terms of two quantities, called the Haussdorff and the Correlation fractal dimension, which are usually significantly smaller than the number of dimensions of the feature space, and effectively deflate the curse of dimensionality.

The mapping from a higher-dimensional to a lower-dimensional space, called *dimensionality reduction*, is normally accomplished through one of three classes of methods: variable-subset selection (possibly following a linear transformation of the space), multidimensional scaling, and geometric hashing.

2.5.1 Variable-Subset Selection

Variable-subset selection consists of retaining some of the dimensions of the feature space and discarding the remaining ones. This class of methods is often used in statistics or in machine learning [25]. In CBIR systems, where the goal is to minimize the error induced by approximating the original vectors with their lower-dimensionality projections, variable-subset selection is often preceded by a linear transformation of the feature space. Almost universally, the linear transformation (a combination of translation and rotation) is chosen so that the rotated features are uncorrelated, or, equivalently, so that the covariance matrix of the transformed dataset is diagonal. Depending on the perspective of the author and on the framework, the method is called Karhunen-Loève transform (KLT) [59, 77], Singular Value Decomposition (SVD) [90], or Principal Component Analysis (PCA) [94, 125] (while the setup and numerical algorithms might differ, all the above methods are essentially equivalent). A variable-subset selection step then discards the dimensions having smaller variance. The rotation of the feature space induced by these methods is optimal in the sense that it minimizes the mean squared error of the approximation resulting from discarding the d' dimensions with smaller variance, for every d'. This implies that, on average, the original vectors are closer (in Euclidean distance) to their projections when the rotation decorrelates the features than with any other rotation.

PCA, KLT and SVD are data-dependent transformations and are computationally expensive. They are therefore poorly suited for dynamic databases where items are added and removed on a regular basis. To address this problem Ravi Kanth, Agrawal and Singh [142] proposed an efficient method for updating the SVD of a dataset, and devised strategies to schedule and trigger the update.

2.5.2 Multidimensional Scaling

Non-linear methods can reduce the dimensionality of the feature space. Numerous authors advocate the use of *multidimensional scaling* [105] for content-based retrieval applications. Multidimensional scaling comes in different flavors, hence it lacks a precise definition. The approach described in [162] consists of remapping the space $\mathbb{R}^n$ into $\mathbb{R}^m$ $(m < n)$ using m transformations each of which is a linear combination of appropriate radial basis functions. This method was adopted by [11] for database image retrieval. The *metric* version of multidimensional scaling [158] starts from the collection of all pairwise distances between the objects of a set, and tries to find the smallest-dimensionality Euclidean space where the objects can be represented as points whose Euclidean distances are "close enough" to the original input distances. Numerous other variants of the method exist.

Faloutsos and Lin [67] proposed an efficient solution to the metric problem, called *FastMap*. The gist of this approach is pretending that the objects are indeed points in an n-dimensional space (where n is large and unknown) and trying to project these unknown points onto a small number of orthogonal directions.

In general, multidimensional scaling algorithms can provide better dimensionality reduction than linear methods, but are computationally much more expensive, modify the metric structure of the space in a fashion that depends on the specific data set, and are poorly suited for dynamic databases.

2.5.3 Geometric Hashing

Geometric Hashing [36, 166] consists of hashing from a high-dimensional space to a very low-dimensional space (the real line or the plane). In general hashing functions are not data-dependent. The metric properties of the hashed space can be significantly different from those of the original space. Additionally, an ideal hashing function should spread the database uniformly across the range of the low-dimensionality space, but the design of such a function becomes increasingly complex with the dimensionality of the original space. Hence, geometric hashing can be applied to image database indexing only when the original space has low-dimensionality, and when only local properties of the metric space need to be maintained.

A few approaches have been proposed that do not fall in any of the three classes described above. An example is the indexing scheme called *Clustering and Singular Value Decomposition* (*CSVD*) [157, 156], in which the index preparation step includes recursively partitioning the observation space into non-overlapping clusters, and applying SVD and variable-subset selection independently to each cluster. Similar approaches have since appeared in the literature, that confirm the conclusions [4, 42]. Aggarwal et al. [4] describes an efficient method for combining the clustering step with the dimensionality reduction, but the paper does not contain applications to indexing. A different decomposition algorithm is described in [42], where the empirical results on indexing performance and behavior are in remarkable

agreement with those in [157, 156].

2.5.4 Some Considerations

Dimensionality reduction allows the use of efficient indexing structures. However, the search is now no longer performed on the original data.

The main downside of dimensionality reduction is that it affects the metric structure of the search space in at least two ways. First, all the mentioned approaches introduce an approximation, which might affect the ranks of the query results. The results of type 2 or 3 queries executed in the original space and in the reduced-dimensionality space need not be the same. This approximation might or might not negatively affect the retrieval performance: since feature-based search is in itself approximate, and since dimensionality reduction partially mitigates the "curse of dimensionality", improvement rather than deterioration is possible. To quantify this effect, experiments measuring precision and recall of the search can be used, where users compare the results retrieved from the original and the reduced dimensionality space. Alternatively, the original space can be used as the reference (in other words, the query results in the original space are used as baseline), and we can measure the difference in retrieval behavior [157].

The second type of alteration of the search space metric structure depends on the individual algorithm. Linear methods, such as SVD (and the non-linear CSVD), use rotations of the feature space. If the same non-rotationally-invariant distance function is used before and after the linear transformation, then the distances between points in the original and in the rotated space will be different, even without accounting for the variable-subset selection step (for instance, when using $D^{(\infty)}$, the distances could vary by a factor of $\sqrt{d}$). However, this problem does not exist when a rotationally invariant distance or similarity index is used. When non-linear multidimensional scaling is used, the metric structure of the search space is modified in a position-dependent fashion, and the problem cannot be mitigated by an appropriate choice of metric.

The methods that can be used to quantify this effect are the same ones proposed to quantify the approximation induced by the dimensionality reduction. In practice, distinguishing between the contributions of the two discussed effects is very difficult and probably of minor interest, and as a consequence a single set of experiments is used to determine the overall combined influence on retrieval performance.

3 Taxonomies of Indexing Structures

After feature selection and dimensionality reduction, the third step in the construction of an index for an image database is the selection of an appropriate indexing structure, a data structure that simplifies the retrieval task. The literature on the topic is immense, and an

exhaustive overview would require an entire book.

Here, we will quickly review the main classes of indexing structures, describe their salient characteristics and discuss how well they can support queries of the three main classes and four categories defined in Section 2.2. The appendix describes in detail the different indexes and compares their variations. This section describes different ways of categorizing indexing structures. A taxonomy of spatial access methods can also be found in [79], which also contains an historical perspective on the evolution of spatial access methods, a description of several indexing methods, and references to comparative studies.

A first distinction, adopted in the rest of the chapter, is between *vector space indexes* and *metric space indexes*. The former represent objects and feature vectors as sets or points in a $d-$dimensional vector space. For example, 2-dimensional objects can be represented as regions of the $x - y$ plane, and color histograms as points in high-dimensional space where each coordinate corresponds to a different bin of the histogram. After embedding the representations in an appropriate space, a convenient distance function is adopted, and indexing structures to support the different types of queries are constructed accordingly. Metric space indexes start from the opposite end of the problem: given the pairwise distances between objects in a set, an appropriate indexing structure is constructed for these distances. The actual representation of the individual objects is immaterial; the index tries to capture the metric structure of the search space.

A second division is algorithmic. We can distinguish between non-hierarchical, recursive partitioning, projection-based, and miscellaneous methods. *Non-hierarchical* schemes divide the search space into regions having the property that the region to which a query point belongs can be identified in a constant number of operations. *Recursive partitioning* methods organize the search space in a way that is well captured by a tree, and try to capitalize on the resulting search efficiency. *Projection-based* approaches, usually well-suited for approximate or probabilistic queries, rely on clever algorithms that perform searches on the projections of database points onto a set of directions.

We can also take an orthogonal approach and divide the indexing schemes into *spatial indexing methods* (SAM), that index spatial objects (lines, polygons, surfaces, solids etc.), and *point access methods* (PAM), that index points in multidimensional spaces. Spatial data structures are extensively analyzed in [148]. Point access methods have been used in pattern recognition applications, especially for nearest-neighbor searches [55]. The distinction between SAMs and PAMs is somewhat fuzzy. On the one hand, numerous schemes exist that can be used as either SAMs or PAMs with very minor changes. On the other, many authors have mapped spatial objects (especially hyperrectangles) into points in higher dimensional spaces, called parameter space [130, 131, 112, 113, 128], and used PAMs to index the parameter space. For example, a $d-$dimensional hyperrectangle aligned with the coordinate axes is uniquely identified by its two vertices lying on its main diagonal, that is, by $2d$ numbers.

4 The Main Classes of Multidimensional Indexing Structures

This section contains a high-level overview of the main classes of multidimensional indexes. We have organized them taxonomically, dividing them into vector-space methods and metric-space methods, and further subdividing each category. The appendix contains a detailed descriptions, discusses individual methods belonging to each subcategory, compares methods within each class and provides references to the immense literature.

4.1 Vector-Space Methods

We divide vector-space approaches into non-hierarchical methods, recursive decomposition approaches, projection-based algorithms, and miscellaneous indexing structures.

4.1.1 Non-Hierarchical Methods

Non-hierarchical methods constitute a wide class of indexing structures. Ignoring the brute-force approach (namely, the sequential scan of the database table), we divide them into two classes.

The first group (described in detail in Appendix A.1) maps the $d-$dimensional spaces onto the real line by means of a space-filling curve (such as the Peano curve, the z-order, and the Hilbert curve), and indexes the mapped records using a one-dimensional indexing structure. Since space-filling curves tend to map nearby points in the original space into nearby points on the real line, range queries, nearest-neighbor queries and $\alpha-$cut queries can be reasonably approximated by executing them in the projected space.

The second group of methods partitions the search space into a predefined number of non-overlapping fixed-size regions, that do not depend on the actual data contained in the database.

4.1.2 Recursive Partitioning Methods

Recursive partitioning methods (see also Appendix B) recursively divide the search space into progressively smaller regions which depend on the dataset being indexed. The resulting hierarchical decomposition can be well-represented by a tree.

The three most commonly used categories of recursive partitioning methods are quadtrees, k-d-trees and R-trees.

Quadtrees divide a $d-$dimensional space into 2^d regions by simultaneously splitting all axes into two parts. Each non-terminal node has therefore 2^d children, and, as in the other

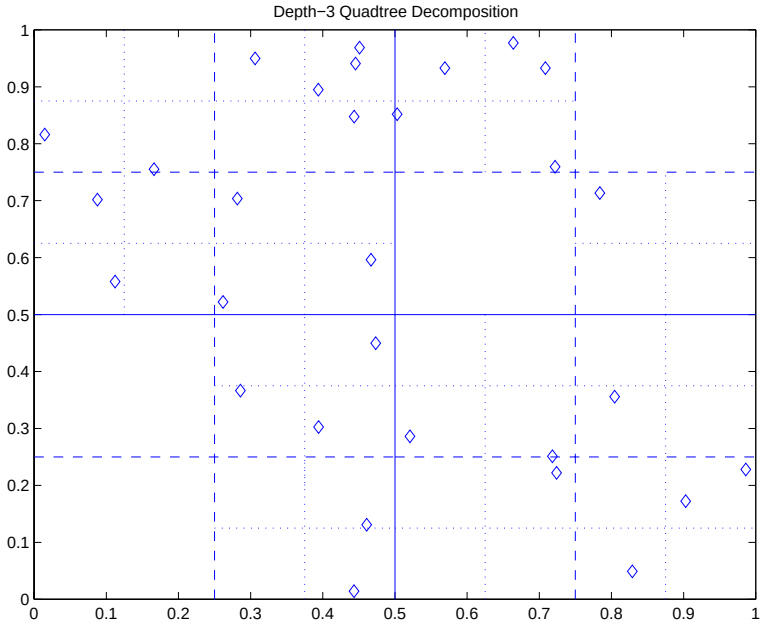

Figure 3: Two-dimensional space decomposition using a depth-3 quadtree. Database vectors are represented as diamonds. Different line types correspond to different levels of the tree. Starting from the root, these line types are: solid, dashed and dotted.

two classes of methods, correspond to hyperrectangles aligned with the coordinate axes. Figure 3 shows a typical quadtree decomposition in a two-dimensional space.

K-d-trees divide the space using $(d-1)-$dimensional hyperplanes perpendicular to a specific coordinate axis. Each non-terminal node has therefore at least two children. The coordinate axis can be selected using a round-robin criterion, or as a function of the properties of the data indexed by the node. Points are stored at the leaves, and, in some variations of the method, at internal nodes. Figure 4 is an example of a k-d-tree decomposition of the same dataset used in Figure 3.

R-trees divide the space into a collection of possibly overlapping hyperrectangles. Each internal node corresponds to a hyperrectangular region of the search space, which generally contain the hyperrectangular regions of the children. The indexed data is stored at the leaf nodes of the tree. Figure 5 shows an example of R-tree decomposition of the same dataset used in Figures 3 and 4. From the figure, it is immediately clear that the hyperrectangles of different nodes need not be disjoint. This adds a further complication that was not present in the previous two classes of recursive decomposition methods.

Variations of the three types of methods exist that use hyperplanes (or hyperrectangles) having arbitrary orientations, or non-linear surfaces (such as spheres or polygons) as partitioning elements.

Though these methods were originally conceived to support point queries and range queries in low-dimensional spaces, they also support efficient algorithms for $\alpha-$cut and

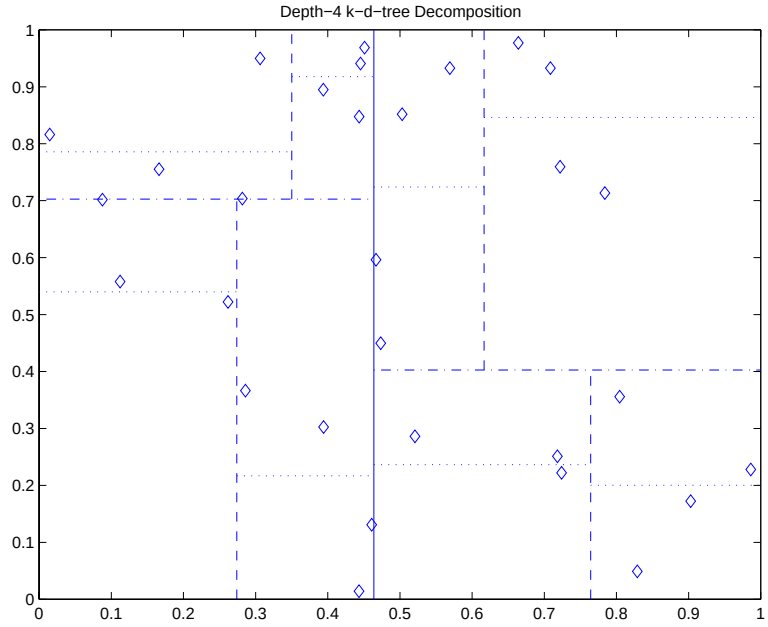

Figure 4: Two-dimensional space decomposition using a depth-4 k-d-b-tree, a variation of the k-d-tree characterized by binary splits. Database vectors are denoted by diamonds. Different line types correspond to different levels of the tree. Starting from the root, these line types are: solid, dash-dot, dashed and dotted. The dataset is identical to that of Figure 3.

nearest-neighbor queries (described in the Appendix).

Recursive-decomposition algorithms have good performance even in 10-dimensional spaces, and can occasionally be useful to index up to 20-dimensions.

4.1.3 Projection-based methods

Projection-based methods are indexing structures that support approximate nearest-neighbor queries. They can be further divided into two categories, corresponding to the type of approximation performed.

The first subcategory, described in Appendix C.1, supports fixed-radius queries. Several methods project the database onto the coordinate axes, maintain a list for each collection of projections, and use the list to quickly identify a region of the search space containing a hypersphere of radius r centered on the query point. Other methods project the database onto appropriate $(d+1)-$dimensional hyperplanes, and find nearest neighbors by tracing an appropriate line[3] query point and finding its intersection with the hyperspaces.

The second subcategory, described in Appendix C.2, supports $(1+\epsilon)$-nearest-neighbor queries, and contains methods that project high-dimensional databases onto appropriately

[3]Details on what constitutes an appropriate line are contained in Appendix C.1.

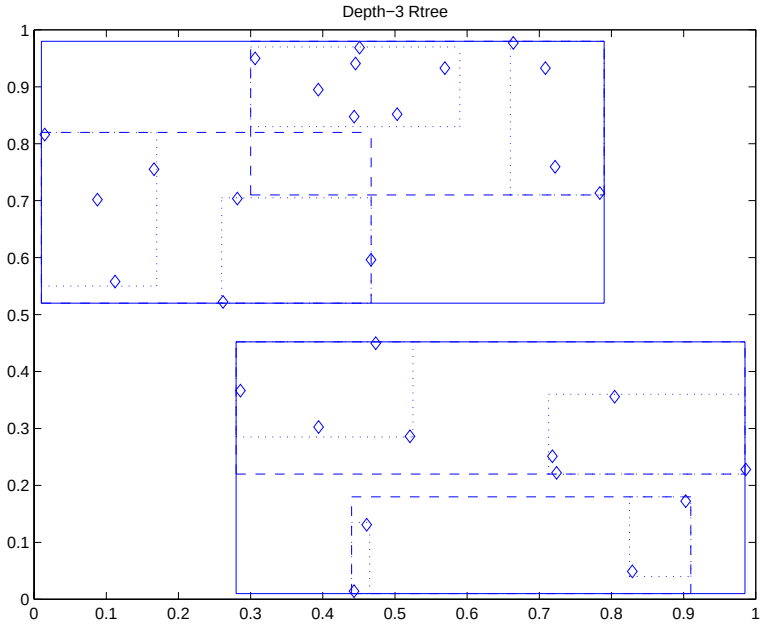

Figure 5: Two-dimensional space decomposition using a depth-3 R-tree. The dataset is identical to that of Figure 3. Database vectors are represented as diamonds. Different line types correspond to different levels of the tree. Starting from the root, these line types are: solid, dashed and dotted.

selected or randomly generated lines, and index the projections. Although probabilistic and approximate in nature, these algorithms support queries whose cost grows only linearly in the dimensionality of the search space and are therefore well-suited for high-dimensional spaces.

4.1.4 Miscellaneous Partitioning Methods

There are several methods that do not fall into any of the previous categories. Appendix C.2 describes three of them: CSVD, the Onion index, and Berchtold', Böhm', and Kriegel's Pyramid (not to be confused with the homonymous quadtree-like method described in Appendix B.1).

CSVD recursively partitions the space into "clusters", and independently reduces the dimensionality of each using Singular Value Decomposition. Branch-and-bound algorithms exist to perform approximate nearest-neighbor and $\alpha-$cut queries. Medium- to high-dimensional natural data, such as texture vectors, appear to be well-indexed by CSVD.

The Onion index indexes a database by recursively constructing the convex hull of its points and "peeling it off". The data is hence divided into nested layers, each of which consist of the convex hull of the contained points. The onion index is well suited for search problems where the database items are scored using a convex scoring function (for instance, a linear

function of the feature values), and the user wishes to retrieves the k items with highest score, or all the items with score exceeding a threshold. We immediately note a similarity with $k-$nearest-neighbor and $\alpha-$cut queries; the difference is that $k-$nearest-neighbor and $\alpha - cut$ queries usually seek to maximize a concave rather than a convex scoring function.

The Pyramid divides the $d-$dimensional space into $2d$ pyramids centered at the origin and with heights aligned with the coordinate axes. Each pyramid is then sliced by $(d-1)-$dimensional equidistant hyperplanes, perpendicular to the coordinate axes. Algorithms exist to perform range queries.

4.2 Metric-Space Methods

Metric-space methods index the distances between database items rather than the individual database items. They are useful when the distances are provided with the dataset (for example, as result of psychological experiments), or where the selected metric is too computationally complex for interactive retrieval (and therefore it is more convenient to compute pairwise distances while adding items to the database).

Most metric-space methods are tailored towards solving nearest-neighbor queries, and are not well-suited for $\alpha-$cut queries. Few metric-space methods have been specifically developed to support $\alpha-$cut queries, but are not well-suited for nearest-neighbor searches. In general, metric-space indexes do not support range queries[4].

We can distinguish two main classes of approaches: those that index the metric structure of the search space and those that rely on vantage points.

4.2.1 Indexing the Metric Structure of a Space

There are two main ways of indexing the metric structure of a space to perform nearest-neighbor queries. The first is applicable when the distance function is known, and consists of *indexing the Voronoi regions* of each database item. Given a database, we can associate with each point of the feature space the closest database item. The collection of feature space points associated with a database item is called its Voronoi region. Different distance functions produce different sets of Voronoi regions. An example of this class of indexes is the *cell* method [19] (Appendix A), which approximate Voronoi regions by means of their minimum-bounding rectangles (MBR), and indexes the MBRs with a X-tree [20] (Appendix B.3).

The second approach is viable when all the pairwise distances between database items are given. In principle, then, it is possible to associate with each database item an *ordered list* of all the other items, sorted in ascending order of distance. Nearest-neighbor queries

[4]It is worth recalling that algorithms exist to perform all the three main similarity query types on each of the main recursive-partitioning vector-space indexes.

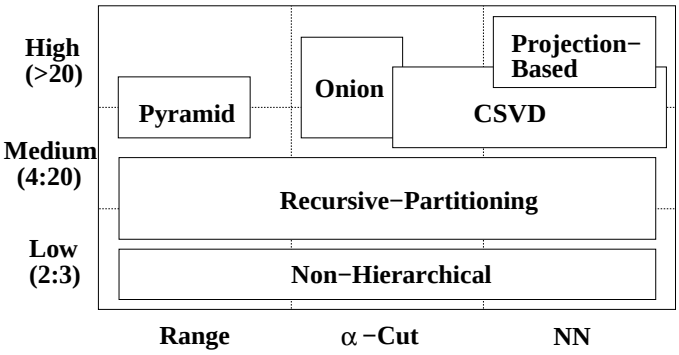

Figure 6: Selecting vector-space methods by dimensionality of the search space and query type.

are then reduced to a point query followed by walking the list. Methods of this category are variations of this basic scheme, and try to reduce the complexity of constructing and maintaining the index.

4.2.2 Vantage-Point Methods

Vantage-point methods (Appendix B rely on a tree structure to search the space. The vp-tree is a typical example of this class of methods. Each internal node indexes a disjoint subset of the database, has two children, and is associated with a database item called the *vantage point*. The items indexed by an internal node are sorted in increasing distance from the vantage point, the median distance is computed, the items closer to the vantage point than the median distance are associated with the left subtree and the remaining ones with the right subtree. The indexing structure is well-suited for fixed-radius nearest-neighbor queries.

5 Choosing an Appropriate Indexing Structure

It is very difficult to select an appropriate method for a specific application. There is currently no recipe to decide which indexing structure to adopt. In this section we provide very general data-centric guidelines to narrow the decision to a few categories of methods.

The characteristics of the data and the metric used dictate whether it is most convenient to represent the database items as points in a vector space or to index the metric structure of the space.

The *useful dimensionality* is the other essential characteristics of the data. If we require exact answers, the useful dimensionality is the same as the original dimensionality of the dataset. If approximate answers are allowed and dimensionality-reduction techniques can be used, then the useful dimensionality depends on the specific database and on the tolerance to approximations (specified, for example, as the allowed region in the precision-recall space). Here, we (somewhat arbitrarily) distinguish between low-dimensional spaces (with 2 or 3

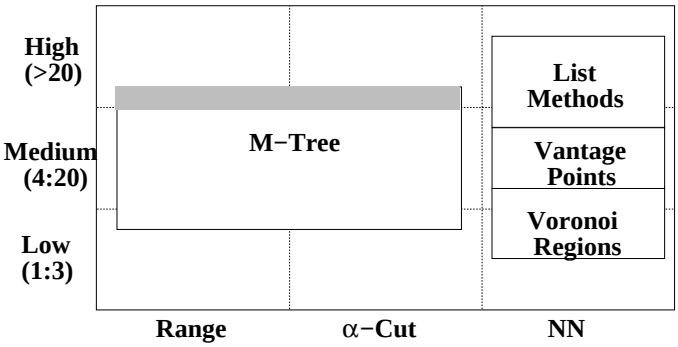

Figure 7: Selecting metric-space methods by dimensionality of the search space and type of query.

dimensions), medium-dimensional spaces (with 4 to 20 dimensions) and high-dimensional spaces, and use this categorization to guide our selection criterion.

Finally, we will select a category of methods that supports the desired type of query (range, $\alpha-$cut or nearest-neighbor).

Figure 6 provides rough guidelines to selecting vector-space methods given the dimensionality of the search space and the type of query. Non-hierarchical methods are in general well-suited for low-dimensionality spaces, and algorithms exist to perform the three main types of queries. In general their performance decays very quickly with the number of dimensions. Recursive-partitioning indexes perform well in low- and medium-dimensionality spaces, they are designed for point and range queries, and the Appendix describe algorithms to perform nearest-neighbor queries, which can also be adapted to $\alpha-$cut queries. CSVD can often capture well the distribution of natural data, and can be used for nearest-neighbor and $\alpha-$cut queries in up to 100 dimensions, but not for range queries. The Pyramid technique can be used to cover this gap, although it does not gracefully support nearest-neighbor and $\alpha-$cut queries in high dimensions. The Onion index supports a special case of $\alpha-$cut queries (where the score is computed using a convex function). Projection-based methods are well-suited for nearest-neighbor queries in high-dimensional spaces, however their complexity does not make them competitive with recursive-partitioning indexes in less than 20 dimensions.

Figure 7 guides the selection of metric-space methods, the vast majority of which support nearest-neighbor searches. A specific method, called the M-tree (Appendix D) can support range and $\alpha-$cut searches in low- and medium-dimensionality spaces, but is a poor choice for high-dimensional spaces. The remaining methods are only useful for nearest-neighbor searches. List methods can be used in medium-to-high dimensional spaces, but their complexity precludes their use in low-dimensional spaces. Indexing Voronoi regions is a good solution to the $1-$nearest-neighbor search problem except in high-dimensionality spaces. Vantage point methods are well-suited for medium-dimensionality spaces.

Once a few large classes of candidate indexing structures have been identified, the other

constraints of the problem can be used to further narrow the selection. We can ask whether probabilistic queries are allowed, whether there are space requirements, limits on the pre-processing cost, constraint on dynamically updating the database, etc. The appendix details this information for numerous specific indexing schemes.

The class of recursive-partitioning methods is especially large. Often structures and algorithms have been developed to suit specific characteristics of the datasets, which are difficult to summarize, but are described in detail in the appendix.

5.1 A Caveat

Comparing indexing methods based on experiments is always extremely difficult. The main problem, is of course, the data. Almost invariably, the performance of an indexing method on real data is significantly different from the performance on synthetic data, sometimes by almost an order of magnitude. Extending conclusions obtained on synthetic data to real data is therefore questionable. On the other hand, due to the lack of an established collection of benchmarks for multidimensional indexes, each author performs experiments on data at hand, which makes it difficult to generalize the conclusions. Theoretical analysis is often tailored towards worst-case performance or probabilistic worst-case performance, more rarely to average performance. Unfortunately, it also appears that some of the most commonly used methods are extremely difficult to analyze theoretically.

6 Future Directions

In spite of the large body of literature, the field of multidimensional indexing appears to still be very active. Aside from the everlasting quest for newer, better indexing structures, there appear to be at least three new directions for research, which are especially important for image databases.

In image databases, often the search is based on a combination of heterogeneous types of features (i.e., both numeric and categorical), specified at query-formulation time. Traditional multidimensional indexes do not readily support this type of query.

Iterative refinement is an increasingly popular way of dealing with the approximate nature of query specification in multimedia databases. The indexing structured described in this chapter are not well-suited to support iterative refinements, except when a query-rewrite strategy is used, namely, when the system does not keep a history of the previous refinement iterations.

Finally, it is interesting to note that the vast majority of the multidimensional indexing structures are designed and optimized for obsolete computer architectures. There seems to be a consensus that most of the index should reside on disk and that only a part of it should

be cached in main memory. This view made sense in the mid-eighties, when a typical 1 MIPS machine would have 64-128 KB of RAM (almost as fast as the processor). In the meantime, several changes have occurred: the speed of the processor has increased by three orders of magnitude (and dual-processor PC-class machines are very common), the amount of RAM has increased by four orders of magnitude, and the size of disks has increased by five or six orders of magnitude. At the same time, the gap in the speed between processor and RAM has become increasingly wide, prompting the need for multiple levels of cache, while the speed of disks has barely tripled. Accessing a disk is essentially as expensive today as it was 15 years ago. However, if we think of accessing a processor register as opening a drawer of our desk to get an item, accessing a disk is the equivalent of going from New York to Sidney to retrieve the same information (though latency hiding techniques exist in multitasking environments). Systems supporting multimedia databases are now sized in such a way that the indexes can comfortably reside in main memory, while the disks contain the bulk of the data (images, video clips etc.) Hence, metrics such as the average number of pages accessed during a query are nowadays of lesser importance. The concept of a page itself is not well-suited to current computer architectures, where performance is strictly related to how well the memory hierarchy is used. Cache-savvy algorithms can potentially be significantly faster than similar methods that are oblivious to memory hierarchy.

References

[1] D.J. Abel and J.L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision, Graphics and Image Processing*, 27(1), 1983.

[2] P.K. Agarwal and J. Matoušek. Ray shooting and parametric search. In *Proc. 24th ACM Symp. Theory of Computing (STOC)*, pages 517–526, 1992.

[3] C. Aggarwal, J.L. Wolf, P.S. Yu, and M. Epelman. The S-Tree: an efficient index for multidimensional objects. In *Proc. Symp. Large Spatial Databases (SSD)*, volume 1262, Lecture Notes in Computer Science, pages 350–373, Berlin, Germany, 1997. Springer.

[4] C.C. Aggarwal, J.L. Wolf, P.S. Yu, Procopiuc C., and Park J.S. Fast algorithms for projected clustering. In *Proc. 1999 ACM SIGMOD Int. Conf. on Management of Data*, pages 61–72, Philadelphia, PA, USA, June 1999.

[5] W.G. Aref and H. Samet. Efficient processing of window queries in the pyramid data structure. In *Proc. ninth ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 265–272, 1990.

[6] W.G. Aref and H. Samet. Optimization strategies for spatial query processing. In *Proc. 17th Int. Conf. on Very Large Data Bases VLDB '92*, pages 81–90, September 1991.

[7] S. Arya and D.M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. 4th Annual ACM-SIAM Symp. Discrete Algorithms*, pages 271–280. ACM, 1993.

[8] S. Arya, D.M. Mount, N.S. Netanyahu, and R. Silverman. An optimal algorithm for approximate nearest neighbor searches. In *Proc. 5th Annual ACM-SIAM Symp. Discrete Algorithms*, chapter 63, pages 572–582. ACM, New York, 1994.

[9] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, and A.Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM*, 45(6):891–923, November 1998.

[10] F. Aurenhammer. Voronoi diagrams - a survey of fundamental geometric data structures. *ACM Computing Surveys*, 23(3):345–405, September 1991.

[11] M. Beatty and B.S. Manjunath. Dimensionality reduction using multi-dimensional scaling for content-based retrieval. In *Proc. IEEE Int. Conf. Image Processing, ICIP '97*, volume 2, pages 835–838, Santa Barbara, CA, October 1997.

[12] N. Beckmann, H.P. Kriegel, R. Shneider, and B. Seeker. The R*−tree: an efficient and robust access method for points and rectangles. In *Proc. 1990 ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, Atlantic City, NJ, USA, May 23-25 1990.

[13] K. Bennett, U. Fayyad, and Geiger D. Density-based indexing for approximate nearest-neighbor queries. In *Proc. 5th ACM SIGKDD Int. Conf. Knowledge Discovery & Data Mining*, pages 233–243, San Diego, CA, August 15-18 1999.

[14] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[15] J.L. Bentley. Multidimensional binary search in database applications. *IEEE Trans. Software Engineering*, 4(5):333–340, 1979.

[16] J.L. Bentley and J.H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, 1997.

[17] S. Berchtold, C. Bohm, B. Braunmueller, D. A. Keim, and H.-P. Kriegel. Fast parallel similarity search in multimedia databases. In *Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data*, pages 1–12, Tucson, AZ, 12-15 May 1997.

[18] S. Berchtold, C. Böhm, and H.-P. Kriegel. The pyramid-tree: Breaking the curse of dimensionality. In *Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data*, pages 142–153, 1998.

[19] S. Berchtold, B. Erlt, D.A. Keim, H.-P. Kriegel, and T. Seidl. Fast nearest neighbor search in high-dimensional space. In *Proc. 14th Int. Conf. on Data Engineering*, pages 209–218, 1998.

[20] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proc. 19th Int. Conf. on Very Large Data Bases VLDB '93*, pages 28–39, Bombay, India, September 1996.

[21] S. Berchtold and H.-P. Kriegel. S3: Similarity search in cad database systems. In *Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data*, pages 564–567, Tucson, AZ, 12-15 May 1997.

[22] L. D. Bergman and V. Castelli. The match score image: A visualization tool for image query refinement. In *Proc. SPIE*, volume 3298, *Visual Data Explor. Anal. V*, pages 172–183, May 1998.

[23] M. Bern. Approximate closest-point queries in high dimensions. *Inf. Proc. Lett.*, 45:95–99, 1993.

[24] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *Proc. Int. Conf. Database Theory (ICDT '99)*, pages 217–235, Jerusalem, Israel, 1999.

[25] B.V. Bonnlander and A.S. Weigend. Selecting input variables using mutual information and nonparametric density estimation. In *Proc. Int. Symp. Artificial Neural Networks, ISANN94*, pages 42–50, Tainan, Taiwan, 15-17 Dec. 1994.

[26] A. Borodin, R. Ostrovsky, and Y. Rabani. Lower bounds for high dimensional nearest neighbor search and related problems. In *Proc. 31st ACM Symp. Theory of Computing (STOC)*, pages 312–321, Atlanta, Georgia, May 1-10 1999.

[27] T. Bozkaya and M. Ozsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Trans. Database Systems (TODS)*, 24(3):361–404, September 1999.

[28] T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger. GeneSys: a system for efficient spatial query processing. In *Proc. 1994 ACM SIGMOD Int. Conf. on Management of Data*, page 519, 1994.

[29] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proc. 1993 ACM SIGMOD Int. Conf. on Management of Data*, pages 237–246, 1993.

[30] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Parallel processing of spatial joins using R-trees. In *Proc. 12th Int. Conf. on Data Engineering*, pages 258–265, 1996.

[31] C. Buckley, A. Singhal, M. Mitra, and G. Salton. New information retrieval approaches using SMART. In *Proc. 4th Text Retrieval Conf.*, page . National Institute of Standards and Technology, 1995.

[32] W.A. Burkhard. Interpolation-based index maintenance. In *Proc. 2nd ACM SIGACT-SIGMOD Symp. Principles of Database Systems*, pages 76–89, 1983.

[33] W.A. Burkhard. Index maintenance for non-uniform record distribution. In *Proc. 3rd ACM SIGACT-SIGMOD Symp. Principles of Database Systems*, pages 173–180, Waterloo, Canada, 1984.

[34] W.A. Burkhard and R.M. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 16:230–236, 1973.

[35] C.-S. Li et al. Comparing texture feature sets for retrieving core images in petroleum applications. In *Proc. SPIE*, volume 3656, *Storage Retrieval Image Video Datab. VII*, pages 2–11, January 1999.

[36] A. Califano and R. Mohan. Multidimensional indexing for recognizing visual shapes. In *Proc. IEEE Computer Vision and Pattern Recognition, CVPR '91*, pages 28–34, June 91.

[37] Chad Carson, Serge Belongie, Hayit Greenspan, and Jitendra Malik. Region-based image query. In *Proc. IEEE CVPR '97 Workshop on Content-Based Access of Image and Video Libraries*, Santa Barbara, CA, USA, 1997.

[38] G.-H. Cha and C.-W. Chung. H-G tree: and index structure for multimedia database. In *Proc. 3rd IEEE Int. Conf. Multimedia Computing and Systems*, pages 449–452, June 1996.

[39] G.-H. Cha and C.-W. Chung. Multi-mode indices for effective image retrieval in multimedia systems. In *Proc. IEEE Int. Conf. Multimedia Computing and Systems*, pages 152–158, Austin, TX, USA, 28 June - 1 July 1998.

[40] G.-H. Cha and C.-W. Chung. A new indexing scheme for content-based image retrieval. *Multimedia Tools and Applications*, 6(3):263–288, May 1998.

[41] A. Chakrabarti, B. Chazelle, B. Gum, and A. Lvov. A lower bound on the complexity of approximate nearest neighbor searching on the hamming cube. In *Proc. 31st ACM Symp. Theory of Computing (STOC)*, pages 305–311, Atlanta, Georgia, May 1-10 1999.

[42] K. Chakrabarti and S. Mehrotra. Local dimensionality reduction: A new approach to indexing high dimensional spaces. In *Proc. 26th Int. Conf. on Very Large Data Bases VLDB '98*, pages 89–100, Cairo, Egipt, September 2000.

[43] T.M. Chan. Approximate nearest neighbor queries revisited. In *Proc. 13th ACM Symp. Computational Geometry*, pages 352–358, 1997.

[44] T.M. Chan. Approximate nearest neighbor queries revisited. *Discrete & Computational Geometry*, 20:359–373, 1998.

[45] Y.-C. Chang, L.D. Bergman, V. Castelli, J.R. Smith, and C.S. Li. The onion technique: Indexing for linear optimization queries. In *Proc. 2000 ACM SIGMOD Int. Conf. on Management of Data*, pages 391–402, 2000.

[46] V. S. Cherkassky, J. H. Friedman, and H. Wechsler. *From Statistics To Neural Networks: Theory and Pattern Recognition Applications*. Springer-Verlag, 1993.

[47] P. Ciaccia, A. Nanni, and M. Patella. A query-sensitive cost model for similarity queries with M-tree. In *Australasian Database Conf.*, pages 65–76, 1999.

[48] P. Ciaccia, M. Patella, F. Rabitti, and P. Zezula. Indexing metric spaces with M-tree. In *SEBD*, pages 67–86, 1997.

[49] P. Ciaccia, M. Patella, and P. Zezula. M-tree, an efficient access method for similarity search in metric spaces. In *Proc. 23rd Int. Conf. on Very Large Data Bases VLDB '97*, pages 426–435, Athens, Greece, 1997.

[50] K. Clarkson. A randomized algorithm for closed-point queries. *SIAM Journal on Computing*, pages 160–164, 1988.

[51] K. L. Clarkson. An algorithm for approximate closest-point queries. In *Proc. 10th Annual Symp. Computational Geometry*, pages 160–164, 1994.

[52] K. L. Clarkson. Nearest neighbor queries in metric spaces. In *Proc. 29th Symp. Theory Comput.*, pages 609–617, 1997.

[53] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–, 1979.

[54] T.M. Cover and P Hart. Nearest neighbor pattern classification. *IEEE Trans. Information Theory*, IT-13(1):21–27, January 1967.

[55] B.V. Dasarathy, editor. *Nearest Neighbor Pattern Classification Techniques*. IEEE Computer Society, 1991.

[56] I. Daubechies. *Ten Lectures on Wavelets*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1992.

[57] A.P. Dempster, N.M. Laird, and D.B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *J. Royal Statistical Soc. B*, 39(1):1–38, 1977.

[58] L. Devroye, L. Györfi, and G. Lugosi. *A probabilistic theory of pattern recognition*. Springer, New York, 1996.

[59] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. John Wiley & Sons, 1973.

[60] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, 1987.

[61] L.D. Bergman et al. PetroSPIRE: A multi-modal content-based retrieval system for petroleum applications. In *Proc. SPIE*, volume 3846, *Multimedia Storage and Archiving Systems IV* , 1999.

[62] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong. Extendible hashing a fast access method for dynamic files. *ACM Trans. Database Systems (TODS)*, 4(3):315–344, 1979.

[63] C. Faloutsos. Gray codes for partial match and range queries. *IEEE Trans. Software Engineering*, 14(10):1381–1393, October 1988.

[64] C. Faloutsos and P. Bhagwat. Declustering using fractals. *PDIS J. Parallel and Distributed Information Systems*, pages 18–25, 1993.

[65] C. Faloutsos, W. Equitz, M. Flickner, W. Niblack, D. Petkovic, and R. Barber. Efficent and effective querying by image content. *J. Intelligent Information Systems*, 3(3/4):231–262, July 1994.

[66] C. Faloutsos, H.V. Jagadish, and Y. Manolopoulos. Analysis of the n-dimensional quadtree decomposition for arbitrary hyperrectangles. *IEEE Trans. Knowledge and Data Engineering*, 9(3):373–383, May/June 1997.

[67] C. Faloutsos and K.-I. Lin. FastMap: a fast algorithm for indexing, data-mining, and visualization of traditional and multimedia datasets. In *Proc. 1995 ACM SIGMOD Int. Conf. on Management of Data*, pages 163–174, San Jose, CA, May 1995.

[68] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proc. 8th ACM ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems, PODS '89*, pages 247–252, Philadelphia, PA, USA, March 29-31 1989.

[69] R.A. Finkel and L.J. Bentley. Quad trees, a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[70] P. Flajolet. On the performance evaluation of extendible hashing and trie searching. *Acta Informatica*, 20:345–369, 1983.

[71] P. Flajolet, G. Gonnet, C. Puech, and J.M. Robson. The analysis of multidimensional searching in quad-trees. In *Proc. 2nd ACM-SIAM Symp. Discrete algorithms*, pages 100–109, 1991.

[72] Michael Freeston. The BANG file, a new kind of grid file. In *Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data*, pages 260–269, May 1987.

[73] J. Friedman. On bias, variance, 0/1−loss and the curse of dimensionality. *J. Data Mining and Knowledge Discovery*, 1(55), 1997.

[74] J.H. Friedman, F. Baskett, and L.J. Shustek. An algorithm for finding nearest neighbors. *IEEE Trans. Computer*, pages 1000–1006, Oct 1975.

[75] H. Fuchs, G.D. Abram, and B.F. Naylor. On visible surface generation by a priori tre structure. *Computer Graphics*, 14(3):124–133, July 1980.

[76] H. Fuchs, G.D. Abram, and B.F. Naylor. On visible surface generation by a priori tree structure. In *Proc. 7th ACM annual Conf. on Computer Graphics, SIGGRAPH'80*, Boston, MA, USA, 1980.

[77] K. Fukunaga. *Introduction to Statistical Pattern Recognition.* Academic Press, 2nd edition, 1990.

[78] K. Fukunaga and P.M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Trans. Computer*, C-24:750–753, July 1975.

[79] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.

[80] I Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, December 1982.

[81] M.F. Goodchild and A.W. Grandfield. Optimizing raster storage: an examination of four alternatives. In *Proc. Auto-Carto 6*, volume 1, pages 400–407, Ottawa, Canada, October 1983.

[82] D. Greene. An implementation and performance analysis of spatial access data methods. In *Proc. 5th Int. Conf. on Data Engineering*, pages 606–615, Los Angeles, California, February 6-10 1989.

[83] O. Günther and J. Bilmes. Tree based access methods for spatial databases: Implementation and performance evaluation. *IEEE Trans. Knowledge and Data Engineering*, 3(3):342–356, September 1991.

[84] O. Günther and E. Wong. A dual-space representation for geometric data. In *Proc. 13th Int. Conf. on Very Large Data Bases VLDB '87*, pages 501–506, Brighton, U.K., September 1987.

[85] A. Guttman. R-Trees: a dynamic index structure for spatial searching. *ACM SIGMOD Record*, 14(2):47–57, June 1984.

[86] D. Hilbert. Uber die stetige abbildung einer linie auf ein flachenstuck. *Math. Ann.*, 38: , 1891.

[87] K. Hinrichs. Implementation of the grid file: design, concepts and experiences. *BIT*, 25:569–592, 1985.

[88] K. Hinrichs and J. Nievergelt. The grid file: a data structure to support proximity queries on spatial objects. In *Proc. Int. Workshop Graph Theoretic Concepts in Computer Science*, pages 100–113, 1983.

[89] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Systems (TODS)*, 24(2):265–318, June 1999.

[90] R.A. Horn and C.R. Johnson. *Matrix Analysis*. Cambridge University Press, New York, 1992.

[91] A. Hutflesz, H.-W. Six, and P. Widmayer. Twin grid files: space optimizing access schemes. In *Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data*, pages 183–190, Chicago, Il, USA, June 1988.

[92] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proc. 30th ACM Symp. Theory of Computing (STOC)*, pages 604–613, Dallas, Texas, USA, 1998.

[93] H.V. Jagadish. Spatial search with polyhedra. In *Proc. 6th Int. Conf. on Data Engineering*, pages 311–319, February 1990.

[94] I.T. Jolliffe. *Principal Component Analysis*. Springer-Verlag, New York, 1986.

[95] I. Kamel and C. Faloutsos. On packing r-trees. In *Proc. 3rd Int. Conf. Information and Knowledge Management CIKM'93*, pages 490–499, Washington, DC, Nov. 1-5 1993.

[96] I Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proc. 20th Int. Conf. on Very Large Data Bases VLDB '94*, pages 500–509, Santiago, Chile, Sept. 12-15 1994.

[97] B. Kamgar-Parsi and L. Kanal. An improved branch and bound algorithm for computing k-nearest neighbors. *Pattern Recognition Letters*, 3:7–12, 1985.

[98] N. Katayama and S. Satoh. The SR-tree: an index structure for high-dimensional nearest neighbor query. In *Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data*, pages 369–380, Tucson, AZ, 12-15 May 1997.

[99] B.S. Kim and S.B. Park. A fast k nearest neighbor finding algorithm based on the ordered partition. *IEEE Trans. Pattern Analysis and Machine Intelligence*, PAMI-8(6):761–766, November 1986.

[100] J.M Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *Proc. 29th ACM Symp. Theory of Computing (STOC)*, pages 599–607, El Paso, Texas, USA, 1997.

[101] A. Klinger. Pattern and search statistics. In J.S. Rustagi, editor, *Optimizing Methods in Statistics*, pages 303–337. Academic Press, New York, 1971.

[102] D.E. Knuth. *The Art of Computer Programming*, volume 3, Sorting and Searching. Addison Wesley, Reading, MA, 2nd edition, 1973.

[103] C.P. Kolovson and M. Stonebraker. Segment indexes: dynamic indexing techniques for multi-dimensional interval data. In *Proc. 1991 ACM SIGMOD Int. Conf. on Management of Data*, pages 138–147, 1991.

[104] H.P. Kriegel, M. Schiwietz, R. Schneider, and B. Seeger. Performance comparison of point and spatial access methods. In *Proc. 1st Symp. Design Large Spatial Databases*, pages 89–114, 1989.

[105] J.B. Kruskal. Multidimensional scaling. *Psychometrika*, 29(1):1–27, March 1964.

[106] A. Kumar. G-tree: a new data structure for organizing multidimensional data. *IEEE Trans. Knowledge and Data Engineering*, 6(2):341–347, April 1994.

[107] C.-S. Li and V. Castelli. Deriving texture feature set for content-based retrieval of satellite image database. In *Proc. IEEE Int. Conf. Image Processing, ICIP '97*, pages 567–579, Santa Barbara, CA, Oct. 26-29 1997.

[108] K.-I. Lin, H.V. Jagadish, and C. Faloutsos. The TV-tree: and index structure for high-dimensional data. *VLDB Journal*, 3(4):517–542, 1994.

[109] Y. Linde, A. Buzo, and R.M. Gray. An algorithm for vector quantizer design. *IEEE Trans. Communications*, COM-28(1):84–95, January 1980.

[110] C. Liu, A. Ouksel, P. Sistla, N. Rishe, J. Wu, and C. Yu. Performance evaluation of G-tree and its application in fuzzy databases. In *Proc. 5th Int. Conf. Information and Knowledge Management CIKM'98*, pages 235–242, Rockville, MD, USA, 1996.

[111] D. Lomet. DL*−trees. Research Report RC-10860, IBM T.J. Watson Res. Ctr., Yorktown Heights, NY, 1984.

[112] D. Lomet and B. Salzberg. A robust multiattribute search structure. In *Proc. 5th Int. Conf. on Data Engineering*, pages 296–304, Los Angeles, California, February 6-10 1989.

[113] D. Lomet and B. Salzberg. The hB-tree: a multiattribute indexing method with good guaranteed performance. *ACM Trans. Database Systems (TODS)*, 15(4):625–658, December 1990.

[114] Tamminen. M. The extendible cell method for closest point problems. *BIT*, 22:24–41, 1982.

[115] W.Y. Ma and B.S. Manjunath. Edge flow: a framework of boundary detection and image segmentation. In *Proc. IEEE Computer Vision and Pattern Recognition, CVPR '97*, pages 744–749, 1997.

[116] B.S. Manjunath. Image processing in the Alexandria digital library project. In *IEEE ADL*, pages 80–87, 1998.

[117] J. Matoušek. Reporting points in halfspaces. *Computational Geometry: Theory and Applications*, 2:169–186, 1992.

[118] E.M. McCreight. Priority search trees. *SIAM Journal on Computing*, 18(2):257–276, May 1985.

[119] S. Meiser. Point location in arrangements of hyperplanes. *Information and Computation*, 106:286–303, 1993.

[120] H. Mendelson. Analysis of extendible hashing. *IEEE Trans. Software Engineering*, SE-8(6):611–619, 1982.

[121] A. Moffat and J. Zobel. Index organization for multimedia database systems. *ACM Computing Surveys*, 27(4):607–609, December 1995.

[122] G.M. Morton. A computer-oriented geodetic data base and a new technique infile sequencing. Technical report, IBM Ltd., Ottawa, Canada, 1966.

[123] S.A. Nene and S.K. Nayar. Closest point search in high dimensions. In *Proc. IEEE Computer Vision and Pattern Recognition, CVPR '96*, pages 859–865, San Francisco, CA, USA, Jun. 18-20 1996.

[124] S.A. Nene and S.K. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 19(9):989–1003, September 1997.

[125] R. Ng and A. Sedighian. Evaluating multi-dimensional indexing structures for images transformed by principal component analysis. In *Proc. SPIE*, volume 2670, *Storage and Retrieval for Still Image Video Databases*, pages 50–61, San Jose, CA, USA, February 1996.

[126] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, C. Faloutsos, and G. Taubin. The QBIC project: Querying images by content using color texture, and shape. In *Proc. SPIE*, volume 1908, *Storage and Retrieval for Image and Video Databases*, pages 173–187, 1993.

[127] H. Niemann and R. Goppert. An efficient branch-and-bound nearest neighbor classifier. *Pattern Recognition Letters*, 7:67–72, February 1988.

[128] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file, and adaptible, symmetric multi-key file structure. *ACM Trans. Database Systems (TODS)*, 9:38–71, 1984.

[129] Y. Niu, M.T. Özsu, and X Li. 2D-h trees: an indexing scheme for content-based retrieval of images in multimedia systems. In *Proc. 1997 Int. Conf. Intelligent Processing Systems*, pages 1717–1715, Beijin, China, Oct. 28-31 1997.

[130] J. Orenstein. Spatial query processing in an object-oriented database system. In *Proc. 1986 ACM SIGMOD Int. Conf. on Management of Data*, pages 326–336, 1986.

[131] J. Orenstein. A comparison of spatial query processing techniques for native and parameter spaces. In *Proc. 1990 ACM SIGMOD Int. Conf. on Management of Data*, pages 343–352, Atlantic City, NJ, USA, May 23-25 1990. ACM Press.

[132] J. Orenstein and T.H. Merret. A class of data structures for associative searching. In *Proc. 3rd ACM SIGACT-SIGMOD Symp. Principles of Database Systems*, pages 181–190, 1984.

[133] E.J. Otoo. A mapping function for the directory of a multidimensional extendible hashing. In *Proc. 10th Int. Conf. on Very Large Data Bases VLDB '84*, pages 493–506, Singapore, 1984.

[134] E.J. Otoo. Balanced multidimensional extendible hash tree. In *Proc. 5th ACM SIGACT-SIGMOD Symp. Principles of Database Systems*, pages 100–113, 1986.

[135] M. Ouksel. The interpolation-based grid file. In *Proc. 4th ACM SIGACT-SIGMOD Symp. Principles of Database Systems*, pages 20–27, Portland, OR, USA, 1985.

[136] M. Ouksel and P. Scheuermann. Storage mapping for multidimensional linear dynamic hashing. In *Proc. 2nd ACM SIGACT-SIGMOD Symp. Principles of Database Systems*, pages 90–105, 1983.

[137] B.-U. Pagel, F. Korn, and C. Faloutsos. Deflating the dimensionality curse using multiple fractal dimensions. In *Proc. 16th Int. Conf. on Data Engineering*, pages 589–598, San Diego, CA, USA, 2000.

[138] B.-U. Pagel and H.-W. Six. Are window queries representative for arbitrary range queries? In *Proc. 15th ACM ACM Symp. Principles of Database Systems, PODS '96*, pages 150–160, Montreal, Quebec, Canada, 1996.

[139] W. Pennebaker and J. L. Mitchell. *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, New York, 1993.

[140] L. M. Po and C. K. Chan. Tree search structures for image vector quantization. In *Proc. of Int. Symp. Sig. Proc. and Appl. ISSPA 90*, volume 2, pages 527–530, Australia, August 1990.

[141] V. Ramasubramanian and K.K. Paliwal. Fast $k-$dimensional tree algorithms for nearest neighbor search with applications to vector quantization encoding. *IEEE Trans. Signal Processing*, 40(3):518–530, March 1992.

[142] K.V. Ravi Kanth, D. Agrawal, and A.D. Singh. Dimensionality reduction for similarity searching in dynamic databases. In *Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data*, pages 166–176, 1998.

[143] M. Regnier. Analysis of the grid file algorithm. *BIT*, 25:335–357, 1985.

[144] J.T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *Proc. 1981 ACM SIGMOD Int. Conf. on Management of Data*, pages 10–18, May 1981.

[145] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. 1995 ACM SIGMOD Int. Conf. on Management of Data*, pages 71–79, San Jose, California, May 1995.

[146] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proc. 1985 ACM SIGMOD Int. Conf. on Management of Data*, pages 17–31, December 1985.

[147] H. Samet. Quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

[148] Hanan Samet. *The design and analysis of spatial data structures.* Addison-Wesley, 1990.

[149] B. Seeger and Kriegel H.-P. The buddy-tree: an efficient and robust method for spatial data base systems. In *Proc. 16th Int. Conf. on Very Large Data Bases VLDB '92*, pages 590–601, 1990.

[150] B. Seeger and H.-P. Kriegel. Techniques for design and implementation of efficient spatial access methods. In *Proc. 14th Int. Conf. on Very Large Data Bases VLDB '88*, pages 360–371, Los Angeles, CA, USA, 1997.

[151] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *Proc. 13th Int. Conf. on Very Large Data Bases VLDB '87*, pages 507–518, Brighton, England, Sept 1987.

[152] K. Shim, R. Srikant, and R. Agrawal. High-dimensional similarity joins. In *Proc. 13th Int. Conf. on Data Engineering*, pages 301–311, Birmingham, UK, 7-11 April 1997.

[153] J. R. Smith. Integrated spatial and feature image systems: Retrieval analysis and compression. Ph.D. Dissertation, Columbia University, 1997.

[154] N. Strobel, C.-S. Li, and V. Castelli. Texture-based image segmentation and MMAP for digital libraries. In *Proc. IEEE Int. Conf. Image Processing, ICIP '97*, volume I, pages 196–199, Santa Barbara, CA, Oct. 26-29 1997.

[155] W.C. Thibault and Naylor B.F. Set operations on polyhedra using binary space partitioning trees. *Computer Graphics*, 21(4):153–162, July 1987.

[156] A. Thomasian, V. Castelli, and C.-S. Li. Approximate nearest neighbor searching in high-dimensionality spaces the clustering and singular value decomposition method. In *Proc. SPIE*, volume 3527, *Multimedia Storage and Archiving Systems III*, pages 144–154, Boston, MA, Nov. 2-4 1998.

[157] A. Thomasian, V. Castelli, and C.-S. Li. Clustering and singular value decomposition for approximate indexing in high dimensional spaces. In *Proc. 7th Int. Conf. Information and Knowledge Management CIKM'98*, pages 201–207, Bethesda, MD, USA, Nov. 3-7. 1998.

[158] W.S. Torgerson. Multidimensional scaling: I. theory and method. *Psychometrika*, 17:401–419, 1952.

[159] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Overlapping linear quadtrees: a spatio-temporal access method. In *Proc. 6th Int. Symp. Advances in geographic information systems*, pages 1–7, 1998.

[160] J.K. Uhlmann. Metric trees. *Applied Mathematics Letters*, 4(5): , 1991.

[161] J.K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.

[162] Andrew R. Webb. Multidimensional scaling by iterative majorization using radial basis functions. *Pattern Recognition*, 28(5):753–759, 1995.

[163] T. Welch. Bounds on the information retrieval efficiency of static file structures. Technical Report 88, MIT, 1971.

[164] D.A. White and R. Jain. Similarity indexing: Algorithms and performance. In *Proc. SPIE*, volume 2670, *Storage and Retrieval for Still Image Video Databases*, pages 62–73, San Diego, USA, January 1996.

[165] D.A. White and R. Jain. Similarity indexing with the SS-Tree. In *Proc. 12th Int. Conf. on Data Engineering*, pages 516–523, New Orleans, USA, February 1996.

[166] H. Wolfson. Model-based object recognition by geometric hashing. In *Proc. 1st European Conf. Com. Vision*, pages 526–536, April 1990.

[167] Q. Yang, A. Vellaikal, and S. Dao. Mb^+-tree: a new index structure for multimedia databases. In *Proc. IEEE Int. Workshop Multi-Media Database Management Systems*, pages 151–158, Blue Mountain Lake, NY, USA, 28-30 Aug. 1995.

[168] N. Yazdani, M. Ozsoyoglu, and G. Ozsoyoglu. A framework for feature-based indexing for spatial databases. In *Proc. 7th Int. Working Conf Scientific and Statistical Database Management*, pages 259–269, Charlottesville, VA, USA, 28-30 Sept. 1994.

[169] P.N. Yianilos. A dedicated comparator matches symbol strings fast and intelligently. *Electronics Magazine (Cover Story)*, 1983.

[170] P.N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc 2nd Annual ACM-SIAM Symp. Discrete Algorithms*, pages 311–321, 1993.

[171] T.P. Yunck. A technique to identify nearest neighbors. *IEEE Trans. Systems, Man and Cybernetics*, 6(10):678–683, October 1976.

[172] P. Zezula, P. Savino, F. Rabitti, G. Amato, and P. Ciaccia. Processing M-trees with parallel resources. In *Proc. 7th Int. Workshop Research Issues on Data Engineering*, pages 147–154, 1997.

Vector-Space Methods In this appendix we describe non-hierarchical methods, recursive decomposition approaches, projection-based algorithms, and several miscellaneous indexing structures.

A Non-Hierarchical Methods

A significant body of work exists on non-hierarchical indexing methods. The brute-force approach (*sequential scan*), where each record is analyzed in response to a query, belongs to this class of methods. The *inverted list* of Knuth [102] is another simple method, consisting of separately indexing each coordinate in the database. One coordinate is then selected (e.g., the first), and the index is used to identify a set of candidates, which is then exhaustively searched.

We describe in detail two classes of approaches. The first maps a $d-$dimensional space onto the real line through a space-filling curve, the second partitions the space into non-overlapping cells of known size.

Both methods are well-suited to index low-dimensional spaces, where $d \leq 10$, but their efficiency decays exponentially when $d > 20$. Between these two values, the characteristics of the specific datasets determine the suitability of the methods. Numerous other methods exist, such as the BANG file [72], but are not analyzed in detail here.

A.1 Mapping High-Dimensional Spaces onto the Real Line

A class of method exists that addresses multidimensional indexing by mapping the search space onto the real line and then using one-dimensional indexing techniques. The most common approach consists of ordering the database using the positions of the individual items on a space-filling curve [81], such as the *Hilbert* or *Peano-Hilbert curve* [86], or the *z-ordering*, also known as *Morton ordering* [122, 1, 32, 80, 132, 136]. We describe the algorithms introduced in [130] that rely on the z-ordering, as representative. For a description of the *zkdb-tree*, the interested reader is referred to the paper by Orenstein and Merret [132].

The z-ordering works as follows. Consider a database $\boldsymbol{\mathcal{X}}$, and partition the data into two parts, by splitting along the x axis according to a predefined rule (e.g., by dividing positive and negative values of x). The left partition will be identified by the number 0 and the right

by the number 1. Recursively split each partition into two parts, identifying the left part by a 0 and the right part by a 1. This process can be represented as a binary tree, the branches of which are labeled with zeros and ones. Each individual subset obtained through s recursive steps is a strip perpendicular to the x axis, and is uniquely defined by a string of s zeros or ones, corresponding to the path from the root of the binary tree to the node associated with this subset. Now, partition the same database by recursively splitting along the y axis. In this case, a partition is a strip perpendicular to the y axis. We can then represent the intersection of two partitions (one obtained by splitting the x axis and the other obtained by splitting the y axis) by interleaving the corresponding strings of zeros and ones. Note that, if the search space is 2-dimensional, this intersection is a rectangle, while in d dimensions the intersection is a $(d-2)-$dimensional cylinder (that is, a hyperrectangle which is unbounded in $d-2$ dimensions) whose axis is perpendicular to the $x-y$ plane and whose intersection with the $x-y$ plane is a rectangle. The $z-$ordering has several interesting properties. If a rectangle is identified by a string s, it contains all the rectangles whose strings have s as a prefix. Additionally, rectangles whose strings are close in lexicographic order are usually close in the original space, which allows one to perform range and nearest-neighbor queries, as well as spatial joins.

The *HG-tree* of Cha and Chung [38, 39, 40] also belongs to this class. It relies on the Hilbert Curve to map n-dimensional points onto the real line. The indexing structure is similar to a B^*-tree [53]. The directory is constructed and maintained using algorithms that keep the directory coverage to a minimum, and control the correlation between storage utilization and directory coverage.

When the tree is modified, the occupancy of the individual nodes is kept above a minimum, selected to meet requirements on the worst-case performance. Internal nodes consists of pairs (*minimum bounding interval, pointer to child*), where minimum bounding intervals are similar to minimum bounding rectangles, but are not allowed to overlap. In experiments on synthetically generated 4-dimensional datasets containing 100,000 objects, the HG-tree shows improvements on the number of accessed pages of 4 to 25% over the Buddy-Tree [149] on range queries, while on nearest-neighbor queries the best result was a 15% improvement and the worst a 25% degradation.

A.2 Multidimensional Hashing and Grid Files

Grid files [114, 88, 128, 133, 135, 143, 134] are extensions of the *fixed-grid method* [102]. The fixed-grid method partitions the search space into hypercubes of known, fixed size, and group all the records contained in the same hypercube into a bucket. These characteristics make it very easy to identify (for instance, via a table lookup) and search the hypercube that contains a query vector. Well-suited for range queries in small dimensions, fixed grids suffer from poor space utilization in high-dimensional spaces, where most buckets are empty. Grid files attempt to overcome this limitation by relaxing the requirement that the cells be fixed-size hypercubes, and by allowing multiple blocks to share the same bucket, provided

that their union is a hyperrectangle.

The index for the grid file is very simple: it consists of d one-dimensional arrays, called *linear scales*, each of which contains all the splitting points along a specific dimension, and of a set of pointers to the buckets, one for each grid block. The grid file is constructed using a top-down approach, by inserting one record at a time. Split and merge operations are possible during construction and index maintenance. There are two types of split: overflowed buckets are split, usually without any influence on the underlying grid; the grid can also be refined by defining a new splitting point, when an overflowed bucket contains a single grid cell. Merges are possible when a bucket becomes underutilized.

To identify the grid block to which a query point belongs, the linear scales are searched, and the one-dimensional partitions to which an attribute belongs are found. The index of the pointer is then immediately computed and the resulting bucket exhaustively searched. Algorithms for range queries are rather simple, and based on the same principle. Nievergelt, Hinterberger and Sevcik [128] showed how to index spatial objects using grid files, by transforming the $d-$dimensional minimum bounding rectangle into a $2d-$dimensional point. The cost of identifying a specific bucket is $O(d \log n)$, and the size of the directory is linear in the number of dimensions and (in general) superlinear in the database size.

Since the directory size is linear in the number of grid cells, non-uniform distributions that result in most cells being empty adversely affect the space requirement of the index. A solution is to use a hashing function to map data points into their corresponding bucket. *Extendible hashing*, introduced by Fagin, Nievergelt, Pippenger and Strong [62], is a commonly used and widely studied approach [120, 70, 33, 136]. Here we describe a variant due to Otoo [134] (the *BMEH-tree*), suited for higher-dimensional spaces. The index contains a directory and a set of pages. A directory entry corresponds to an individual page and consists of a pointer to the page, a collection of *local depths*, one per dimension, describing the length of the common prefix of all the entries in the page along the corresponding dimension, and a value specifying the dimension along which the directory was last expanded. Given a key, a $d-$dimensional index is quickly constructed that uniquely identifies, through a mapping function, a unique directory entry. The corresponding page can then be searched. A hierarchical directory can be used to mitigate the negative effects of non-uniform data distributions.

G-trees [106, 110] combine B^+-trees [53] with grid-files. The search space is partitioned using a grid of variable size partitions, individual cells are uniquely identified by a string describing the splitting history, and the strings are stored in a B^+-tree. Exact queries and range queries are supported. Experiments in [110] show that, when the dimensionality of the search space is moderate (< 16) and the query returns a significant portion of the database, the method is significantly superior to the Buddy Hash Tree [104] the BANG file [72], the hB-tree [113] (Section B.2), and the 2-level grid file [87]. Its performance is somewhat worse when the number of retrieved items is small.

B Recursive Partitioning Methods

As the name implies, recursive partitioning methods recursively divide the search space into progressively smaller regions, usually mapped into nodes of trees or tries[5], until a termination criterion is satisfied. Most of these methods were originally developed as SAM or PAM, to execute point or range queries in low-dimensionality spaces (typically, for images, geographic information systems applications, and volumetric data), and have subsequently been extended to higher-dimensional spaces. In more recent times, algorithms to perform nearest-neighbor have been proposed for several of them. In this section, we describe three main classes of indexes: quadtrees, k-d-trees and R-trees, which differ in the partitioning step. In each section, we first describe the original method from which all the indexes in the class were derived, then we discuss its limitations and how different variants try to overcome them. For k-d-trees and R-trees a separate subsection is devoted to how to perform nearest-neighbor searches.

We do not describe in detail numerous other similar indexing structures, such as the *range tree* [16], and the *priority search tree* [118].

Note, finally, that recursive partitioning methods were originally developed for low-dimensionality search spaces. It is therefore unsurprising that they all suffer from the curse-of-dimensionality and generally become ineffective when $d > 20$, except in rare cases where the datasets have a peculiar structure.

B.1 Quadtrees and Extensions

Quadtrees [147] are a large class of hierarchical indexing structures, that perform recursive decomposition of the search space. Originally devised to index 2-dimensional data, they have been extended to multidimensional spaces. Three-dimensional quadtrees are called *octrees*; since there is no commonly used name for the d-dimensional extension, we will refer to them simply as quadtrees. Quadtrees are extremely popular in Geographic Information System (GIS) applications, where they are used to index points, lines, and objects.

Typically, quadtrees are trees of degree 2^d, where d is the dimension of the sample space; hence, each non-terminal node has 2^d children. Each step of the decomposition consists of identifying d splitting points (one along each dimension), and partitioning the space by means of $(d-1)$-dimensional hyperplanes passing through the splitting point and orthogonal to the splitting point coordinate axis. Splitting a node of a d-quadtree consists of dividing each dimension into two parts, thus defining 2^d hyperrectangles. A detailed analysis of the retrieval performance of quadtrees can be found in [71].

The various flavors of these indexing structures correspond to the type of data indexed (i.e., points or regions), to specific splitting rules (i.e., strategies for selecting the splitting

[5]With an abuse of terminology, we will not make explicit distinctions between tries and trees, both to simplify the discussion and because the distinction is actually rarely made in the literature.

points) and tree-construction methods, to different search algorithms, and to where the data pointers are contained (just at the leaves or also at intermediate nodes).

The *region quadtree* [101] decomposes the space into squares aligned with the axes, and is well-suited to represent regions. A full region quadtree is also commonly known as a *pyramid*.

The *point quadtree* introduced by Finkel and Bentley [69] use an adaptive decomposition where the splitting points depending on the distribution of the data, and is well-suited to represent points.

While the implementations of region quadtrees and linear quadtrees rely on pointers, more efficient representations are possible. The key observation is that individual nodes of a quadtree can be uniquely identified by strings, called location codes, of 4 symbols, representing the four quadrants (NE, NW,SW,SE). Linear quadtrees [6, 80] rely on location codes to represent the quadtree as an array of nodes, without using pointers.

Extensions to spatiotemporal indexing include the Segment Indexes of Kolovson and Stonebraker [103] (which are really general extensions of multidimensional indexing techniques), and the Overlapping Linear Quadtrees [159].

Point queries (exact searches) using quadtrees are trivial, and are executed by descending the tree until the result is found. Due to the geometry of the space partition, which relies on hyperplanes aligned with the axes, the quadtree can be easily used to perform range queries. In low dimensionality spaces, fast algorithms for range queries exist for quadtrees and pyramids [5].

Quadtrees, however, are not especially well-suited for high-dimensional indexing. Their main drawback is that each split node has always 2^d children, irrespective of the splitting rule. Thus, even in search spaces that are of small dimensions for image databases applications, for instance $d = 20$, the quadtree is in general very sparse, that is, most of its nodes are empty. Faloutsos et al. [66] derive the average number of blocks that need to be visited to satisfy a range query, under the assumption that the conditional distribution of the query hyperrectangle given its dimensions is uniform over the allowable range, and show that it grows exponentially in the number of dimensions. Quadtrees are also rather inefficient for exact $\alpha-$cut and nearest-neighbor queries, since hyperspheres are not well approximated by hyperrectangles (see Section 2.4).

Quadtrees can be used in image databases to index objects segmented from images, and to index features in low-dimensionality spaces. In spite of their simplicity, quadtrees are rarely used for high-dimensional feature indexing, due to their poor performance in high-dimensional spaces.

B.2 K-Dimensional Trees

The k-dimensional tree, known as *k-d tree* [14, 15], is another commonly used hierarchical indexing structure.

The original version, due to Bentley, operates by recursively subdividing the search space one dimension at a time, using a round-robin approach. Each splitting step is essentially identical to that of a binary search tree [102]. Points are stored at both internal nodes and leaves, which somewhat complicates the deletion process. In its original incarnation, the tree is constructed by inserting one point at a time, and the splitting hyperplanes are constrained to pass through one of the points. A subsequent version [16], called the *adaptive k-d-tree*, selects splitting points that divide the number of points in the node into two equal sets.

A dynamic version, called *k-d-b-tree*, supporting efficient insertion and deletion of points, was proposed by Robinson [144]. K-d-b-trees consist of collections of pages: region pages, which correspond to disjoint regions of the search space, and point pages, which contain collections of points. The leaves of the tree are point pages, the internal nodes are region pages. Several splitting strategies, as well as algorithms for efficient insertion, deletion and reorganization, are described in [144]. The method is quite elegant and is amenable to very efficient and compact implementations. As a consequence, it is still commonly used, and is often one of the reference methods in experiments.

A limitation of the k-d-tree is that, in some cases, the index and data nodes cannot be efficiently split, and the result is a significant utilization imbalance. The *hB-trees* (holey brick trees) [113] address this problem by allowing nodes to represent hyperrectangular regions with hyperrectangular holes. This last property makes them also a multidimensional extension of the DL*-tree [111]. The shape of the holey hyperrectangles indexed by an internal node is described by a k-d-tree, called *local k-d-tree*. Several mechanisms for constructing and maintaining the data structure are derived from the $1-$dimensional B$+-$tree [53]. Node splitting during construction is a distinguishing feature of hB-trees: corner splits remove a quadrant from a hyperrectangle or a holey hyperrectangle. Most of the complexity associated with the data structure is concentrated in the splitting algorithms, which guarantees robustness in worst cases. This index supports both exact and range queries.

While most k-d-tree variations split the space along one coordinate at a time, and the partitioning hyperplanes at adjacent levels are perpendicular or parallel, this restriction can be relaxed. For example, the *BSP-tree* (*Binary Space Partition Tree*) [76, 75, 155] uses general partitioning hyperplanes.

Nearest-Neighbor queries using k-d-trees

K-d-trees and variations are well-suited for nearest neighbor searches when the number of indexed dimensions is moderate.

Kim and Park [99] proposed an indexing structure, called the *ordered partition*, which can be considered a variation of the k-d-tree. The database is recursively divided across

individual dimensions, in a round-robin fashion, and the split points are selected to produce subsets of equal size. The recursive decomposition can be represented as a tree, where a node at level ℓ corresponds to a region which is unbounded in the last $d - \ell + 1$ dimensions, and bounded between pair of hyperplanes parallel to the first $\ell - 1$ dimensions. A branch-and-bound search strategy [78, 97] is used to retrieve the k nearest points (in Euclidean distance) to the query sample. First the node to which the query sample belongs (primary node) is identified by descending the tree, and exhaustively searched, thus producing a current k-nearest-neighbor set. The distance of the kth neighbor becomes the bound B. The distances between the query sample and the hyperrectangles of the siblings of the primary node are then computed, and a candidate set is created which includes all nodes at distance less than B. The candidate nodes are exhaustively searched in order of increasing distance from the query. The bound B and the candidate set are updated at the end of every exhaustive search. When all candidates have been visited, a backtracking step is taken by considering the siblings of the parent of the primary node and applying the same algorithm recursively. The search terminates when the root is reached. If the Euclidean distance is used, distance computations can be performed during the traversal of the tree, and require only one subtraction, one multiplication and one addition per node. The method is also suited for general Minkowsky and weighted Minkowsky distances. The authors suggest a fan-out equal to the (average) number of points stored at the leaves. The approach is extremely simple to implement in a very efficient fashion, and produces very good results in few dimensions. However, for synthetic data, the number of visited terminal and non-terminal nodes appears to grow exponentially in the number of dimensions. For real data, the method significantly benefits from a rotation (using SVD) of the feature spaces into uncorrelated coordinates ordered by descending eigenvalues. A similar method was proposed by Niemann and Goppert [127]. The main difference lies in the partitioning algorithm, which is not recursive, and divides each dimension into a fixed number of intervals. A matrix containing the split points along each coordinate is maintained and used during the search. Niemann's approach seems to be twice as fast as previous versions [97].

A data structure that combines properties of quadtrees and of k-d-trees is the *balanced quadtree* [23, 43, 44], which can be used to efficiently perform approximate nearest-neighbor queries provided that the allowed approximation is large. A balanced quadtree, like a k-d-tree, is a binary tree. Coordinates are split in a round-robin fashion, as in the construction of the region quadtree, and, in the same spirit, splits divide the hyperrectangle of inner nodes into (two) hyperrectangles of equal size. To efficiently process nearest-neighbor queries on a database $\boldsymbol{\mathcal{X}}$, a set of vectors $\{\mathbf{v}_j\}$ is considered, and a balanced quadtree is constructed for each translated version $\boldsymbol{\mathcal{X}} + \mathbf{v}_j$ of the database. Several algorithms belonging to this class have been proposed, and are summarized in Table 1. They differ on the number of representative vectors and on the selection criterion for the vectors. They yield different approximations (the ϵ), and have different index construction and search costs.

The *BBD-tree* (balanced-box-decomposition), of Arya, Mount, Netanyahu, Silverman and Wu [8, 9], is an extension that supports $(1+\epsilon)-$approximate nearest-neighbor. The recursive decomposition associates nodes with individual *cells*, each of which is either a hyperrectangle

Author and Reference	Nr. of Vectors	Vector Selection	Value of ϵ	Search Time	Construction Time
Bern [23]	$\sqrt{d}$	Deterministic	2^d	$d \cdot 2^d \cdot \log n$	$d \cdot 8^d \cdot n \cdot \log n$
Bern [23]*	$t \cdot \log n^*$	Randomized	$d^{3/2}$		
Chan [43, 44]	d	Deterministic	$4d^{\frac{3}{2}} + 4d^{\frac{1}{2}} + 1$	$d^2 \log n$	

Table 1: Comparison of $(1+\epsilon)$ nearest neighbors (see Section 2.2 for the definition of this type of queries) relying on balanced quadtrees. As described in the text, this type of searches are supported by constructing a set of vectors, adding each vector to all the database records, and constructing a balanced quadtree. Hence, the index is composed of one balanced quadtree per each vector in the set. The methods summarized differ in the number of required vectors and in the procedure used to select them. These parameters determine the magnitude of the approximation (ϵ), the construction time and the search time. Time values are $O(\cdot)$-order costs; ϵ and the number of vectors are actual values.
$*$ Bern's second algorithm depends on the parameter t: this parameter determines the probability that the algorithm actually finds $(1+\epsilon)$-neighbors (which is equal to $1-O(1/n^t)$), and the number of vectors used in the construction of the index.

Author	1-nn seach time	k-nn seach time	Space requirement	Construction time
Arya et al. [8, 9]	$d \cdot \left(\frac{d}{\epsilon}\right)^d \cdot \log n$	$d \cdot \left(\frac{d}{\epsilon}\right)^d + k \cdot d \cdot \log n$	$d \cdot n$	$d \cdot n \cdot \log n$
Clarkson [51]	$\epsilon^{\frac{1-d}{2}} \cdot \log n$	-	$\epsilon^{\frac{1-d}{2}} \cdot n \cdot \log \rho$	$\epsilon^{1-d} \cdot n^2 \cdot \log \frac{1}{\epsilon}$
Chan [43, 44]	$\epsilon^{\frac{1-d}{2}} \cdot \log n$	-	$\epsilon^{\frac{1-d}{2}} \cdot n \cdot \log n$	$\epsilon^{\frac{1-d}{2}} \cdot n \cdot \log n$

Table 2: The table compares the costs of different algorithms to perform $1+\epsilon$ approximate nearest-neighbor queries using BBD-trees. The values reported are the search times for 1 and k nearest-neighbor queries, the space requirement for the index and the preprocessing time. The reported values describe the $O(\cdot)$-order costs. Recall that d is the number of dimensions, n is the database size, and k is the number of desired neighbors. Clarkson's algorithm is a probabilistic method (guarantees results with high probability) that relies on results in [7]. The quantity ρ is the ratio between the distance between the two furthest points and the distance between the two closest points in the database.

aligned with the axes or the set-theoretic difference between two such nested hyperrectangles. Each leaf contains a single database item. As it is often the case, the tricky part of the tree construction algorithm is the splitting of nodes: here a decision must be made whether to use a hyperplane perpendicular to one of the coordinate axes, or an inner box (called shrinking box). Searching a BBD-tree consists of first finding the cell containing the query point, computing the distances from the other cells, and finally searching the terminal cells in order of increasing distance. A list of current results is maintained during the search. The algorithm terminates when the closest unsearched cell is at a distance larger than $(1-\epsilon)$ times the distance of the farthest current result. The search is therefore approximate, since cells that could contain exact neighbors can be discarded. The decomposition into cells achieves exponential decrease in the number of points and in the volume of the space searched while the tree is visited, and in this sense the BBD-tree combines properties of the optimized k-d-tree and of quadtrees with bounded aspect ratio. Theoretical analysis shows that the number of boxes intersected by a sphere of radius r is $O(r^d)$, thus it grows exponentially in the number of dimensions. The number of leaf cells visited by a k-nearest-neighbor search is $O(k+d^d)$ for any Minkowsky metric, where k is the desired number of neighbors. Experimental results on synthetic datasets containing 100000 points in 15 dimensions, show that, depending on the distribution, the BBD-tree can perform from slightly better than the optimized k-d-tree up to 100 times faster. Additionally, the quality of the search is apparently very good, in spite of the approximation, and the number of missed nearest neighbors is small. Search times, space requirements and construction times for this method and for subsequent improvements are reported in Table 2.

B.3 R-Trees and Derived Methods

The R-tree [85] and its numerous variations (such as [82]) are probably the most studied multidimensional indexing structure, hence we devote a long section to their description. The distinctive feature of the R-tree is the use of hyperrectangles, rather than hyperplanes, to partition the search space. The hyperrectangle associated with a particular node contains all the hyperrectangles of the children.

The most important properties for R-tree performance are overlap, space utilization, rectangle elongation and coverage.

- *Overlap* between k nodes is the area (or volume) contained in at least two of the rectangles associated with the nodes. Overlap for a level is the overlap of all the nodes at that level. If multiple overlapping nodes intersect a query region, they need to be visited, even though only one might contain relevant data.

- *Utilization.* Nodes in a R-tree contain a predefined number of items, often selected to ensure that the size of the node is a multiple of the unit of storage on disk. The utilization of a node is the ratio of the actual number of contained items to its maximum value. Poor utilization affects performance. If numerous nodes have a small number

of descendants, the size of the tree grows, and consequently, the number of nodes to be visited during searches becomes large. Another effect, relevant when the index is paged on disk and only a maximum number of nodes is cached, is the increase in the I/O cost.

- *Node elongation* is the ratio of the lengths of the longest and the shortest sides of the associated hyperrectangle. Bounding rectangles that are very elongated in some directions while thin in others are visited fruitlessly more often than rectangles that are close to squares (or hypercubes). A measure of skewness or elongation is the ratio of surface area to volume, which is minimized by hypercubes.

- *Coverage* is defined for each level of the tree as the total area (or volume) of the rectangles associated with the level. Excessive coverage is a small problem in low dimensions, but it can become significant in high dimensions, where the dataset is sparser and a large part of the covered space is, in fact, empty (dead space).

Since the tree is constructed by inserting one item at a time, the result is highly dependent on the order in which the data is inserted. Different orders of the same data can produce highly optimized trees or very poorly organized trees.

Although some authors, for instance, Faloutsos [65], have reported that the R-tree shows some robustness with respect to the curse-of-dimensionality, being efficient in up to 20 dimensions, over the years there have been numerous extensions of the basic scheme, that attempt to minimize the different causes of inefficiency.

The first class of extensions retain most of the characteristics of the original R-tree. In particular, the partitioning element is a hyperrectangle, data is stored at the leaves, and each data point is indexed by a unique leaf.

R^+-trees [151] address the problem of minimizing overlap. They are constructed in a top-down fashion using splitting rules that generate non-overlapping rectangles. The rectangle associated with each node is constrained to strictly contain the rectangles of its children, if the children are internal nodes, and can overlap the rectangles of its children only if the children are leaves. Copies of a spatial object are stored in each leaf it overlaps. The similarity with k-d-b-trees is not accidental: R^+-trees can, in fact, be thought of as extensions of k-d-b-trees that add two properties: the ability to index spatial objects (rather than points in k-dimensional spaces), and an improvement in coverage, since the coverage of the children of a node need not be equal to the coverage of their parent, and can be significantly smaller. The search algorithms are essentially identical to those used for regular R-trees. The insertion algorithm is a derivation of the downwards split used in the k-d-b-trees. Splitting when a node overflow occurs involves a partitioning and a packing step, which maintain the desired properties of the tree. The R^+-tree shows significant performance improvement over R-trees, especially for large database sizes.

R^*-trees [12] are an attempt to optimize the tree with respect to all four main causes of performance degradation described above. The node selection procedure of the insertion

algorithm selects the node that results in smaller overlap enlargement and resolve ties by minimizing area enlargement. Alternatively, to reduce the high computational cost of the procedure, the nodes are first sorted in increasing value of area enlargement, and the selection algorithm is applied to the first p nodes of the list. When splitting a node, different algorithms can be used that minimize the overall volume increase (area in 2-d), the overall surface area (perimeter in 2-d), the overall overlap, and combinations of the above. A forced reinsertion algorithm is also proposed to mitigate the dependence of the tree on the order in which the data are inserted: during the insertion of a new item, when a split occurs at a particular level, the node is reorganized by selecting an appropriate subset of its children, removing them from the node, and reinserting them from the top. Note that the procedure could induce new splits at the same level from which it was initiated; Forced reinsertion is applied at most once per level during each insertion of a new item. In experiments reported in [12] based on a database containing 100,000 points, the R*-tree outperforms the original R-tree by a factor of 1.7 to 4, and Greene's version [82] by a factor of 1.2 to 2

Packed R-trees are a family of methods that attempt to improve space utilization. The strategy proposed by Roussopoulos and Leifker [146] consists of sorting the data on the one of the coordinates of a selected corner of the minimum bounding rectangle representing the objects. Let n be the maximum number of objects that a leaf of an R-tree can contain. Then the first n items in the sorted list are assigned to the first leaf of the R-tree, the following n to the second leaf and so on. The resulting leaves correspond to rectangles that are elongated in one direction and rather thin in the other. Experimental results suggest that, in 2 dimensions, the approach outperforms quadratic split R-trees and R*-trees for point queries on point data, but not on range queries or for spatial data represented as rectangles. Kamel and Faloutsos introduced a different packed R-tree [95], which overcomes these limitations. Instead of ordering the data according to a chosen dimension, the authors use a space-filling curve, and they select the Hilbert curve over the z-ordering [130], and the Gray-code [63], because results in [68] show that it achieves better clustering than the other two methods. The Hilbert curve induces an ordering on the k-dimensional grid; the authors describe several methods that rely on this ordering to assign a unique numeric index to each of the minimum bounding rectangle (MBR) of the objects. The R-tree is then constructed by first filling the leaves and then building higher level nodes. The first leaf (leaf no1) contains the first n items in the Hilbert order, the second leaf contains the following n items and so on. Higher level nodes are generated by grouping nodes from the previous level in order of creation. For instance, the first node at level 1, which correspond to a set of L leaves, will contain pointers to leaves $1, 2, \ldots, L$. In experiments on two-dimensional real and synthetic data, the number of pages touched by Kamel's and Faloutsos' packed R-tree during range queries is between 1/4 and 1/2 of those touched by Roussopoulos' and Leifker's version, and always outperforms (by a factor of up to 2) the R*-tree and the quadratic split R-tree. The *Hilbert R-tree* [96] is a further development, characterized by sorting hyperrectangles by the Hilbert ordering of their centers, and having intermediate nodes contain information on the Hilbert values of the items stored at its descendant leaves. The Hilbert R-tree has also improved index management properties.

A different approach to minimizing overlap is used in the *X-tree* [20]. Three types of nodes are used: the first two are the data nodes (corresponding to a set of objects, and containing, for each of them, a MBR and a pointer), and the directory nodes (containing, for each child, a MBR and a pointer), have fixed size and are essentially identical to those of a traditional R-tree. The third type of nodes are called supernodes, contain large directories and have variable size. They are used to avoid splits that would produce inefficient directory structures. Appropriate construction and management algorithms ensure that the X-tree maintains minimum node overlap. Experimental comparisons using real data in 2 to 16 dimensions show that insertion is up to 10 time faster than with R*-trees. When performing nearest-neighbor queries on synthetic data, the R*-trees require up to twice the CPU of the X-trees, and access up to 6 times more leaf nodes. When operating on real data, the X-trees are up to 250 times faster than the corresponding R*-trees. When performing point queries on a database of 25000 synthetically generated data points in up to 10 dimensions, the X-tree is essentially equivalent to the TV-Tree [108] (described later), but becomes significantly faster in more than 10 dimensions. Berchtold [17] also proposed a parallel version of the X-tree, where the search space is recursively decomposed into quadrants and a graph coloring algorithm is used to assign different portions of the database to different disks for storage and to different processors during nearest neighbor search. Experimental results, on 8 to 15 dimensional real and artificial data. show that declustering using the Hilbert curve [64] (a well-known competitive approach) is slower than the parallel X-Tree, typically by a factor of $a(n_{disks} - 2) + b$, where n_{disks} is the number of available disks, b varies between 1 and 2, and a is between .3 and .6.

The *VAMSplit R-Tree* [164] optimizes the R-tree construction by employing techniques derived from the k-d-tree. The main gist of this top-down approach is the recursive partitioning of the data space using a hyperplane perpendicular to the coordinate along which the variance of the data is maximum. The split-point selection algorithm minimizes the number of disk blocks required to store the information. In experiments, it appears to outperform both SS-trees and R*-trees.

The *S-Tree* [3] (skew tree) is a data structure for range queries and point queries that tries to minimize the number of pages touched during the search. It combines properties of the R-tree and of the B-Tree [53]. Leaves and internal nodes are identical to the data and directory nodes of the R-tree. Each internal node has a fanout exactly equal to its maximum value, except for "penultimate" nodes, whose children are all leaves. Two parameters of the index are the skew factor p, taking values between 0 and 1/2, and the maximum allowed overlap. Given any pair of sibling nodes, and calling $N1$ and $N2$ the number of leaves in the subtrees rooted at these nodes, the index construction and maintenance try to minimize the total coverage while guaranteeing that $p \leq N1/N2 \leq p^{-1}$ and that the allowable overlap is not exceeded. The index construction time is $O(d \cdot n \cdot \log n)$, where d is the number of dimensions, and n is the database size. Experiments on 2-dimensional synthetic data showed that the method performs better than Hilbert R-tree, but we do not know results in higher dimensions.

Let $\mathbf{x}$ be the query point and R be a hyperrectangle. Let $\mathbf{s}$ and $\mathbf{t}$ be the vertices on the main diagonal of R, with the convention that $\mathbf{s}[i] \leq \mathbf{t}[i]$, $\forall i$.

$$\begin{aligned} \text{MINDIST}(\mathbf{x}, R) &\triangleq \sum_{i=1}^{d} (\mathbf{x}[i] - \mathbf{r}[i])^2 , \\ \text{MINMAXDIST}(\mathbf{x}, R) &\triangleq \min_k \left\{ (\mathbf{x}[k] - \mathbf{rm}[k])^2 + \sum_{i=1,\, i \neq k}^{d} (\mathbf{x}[i] - \mathbf{rM}[i])^2 \right\}, \end{aligned}$$

where

$$\begin{aligned} \mathbf{r}[i] &= \mathbf{x}[i] \quad \texttt{if} \quad \mathbf{s}[i] \leq \mathbf{x}[i] \leq \mathbf{t}[i], \\ &= \mathbf{s}[i] \quad \texttt{if} \quad \mathbf{x}[i] \leq \mathbf{s}[i], \\ &= \mathbf{t}[i] \quad \texttt{if} \quad \mathbf{x}[i] \geq \mathbf{t}[i]. \end{aligned}$$

$$\begin{aligned} \mathbf{rm}[k] &= \mathbf{s}[k] \quad \texttt{if } \mathbf{x}[k] \leq (\mathbf{s}[k] + \mathbf{t}[k])/2, \\ &= \mathbf{t}[k] \quad \texttt{otherwise.} \end{aligned}$$

$$\begin{aligned} \mathbf{rM}[i] &= \mathbf{s}[i] \quad \texttt{if } \mathbf{x}[i] \geq (\mathbf{s}[k] + \mathbf{t}[k])/2, \\ &= \mathbf{t}[i] \quad \texttt{otherwise.} \end{aligned}$$

Table 3: Definition of the metrics MINDIST and MINMAXDIST used by branch-and-bound nearest-neighbor search algorithms using R-trees.

Nearest-Neighbor Queries using R-trees

While range queries (both for overlap and for containment) are easily performed using R-trees, and spatial joins can be performed by transforming them into containment queries [29], the first method for nearest-neighbor queries was proposed by Roussopoulos, Kelley and Vincent [145]. The search relies on two metrics, called MINDIST and MINMAXDIST, defined in Table 3, which measure the distance between a point $\mathbf{x}$ and a hyperrectangle R using the two vertices on the main diagonal, $\mathbf{s}$ and $\mathbf{t}$. MBR's whose MINDIST is larger than the distance to the farthest current result will be pruned, and MBR's whose MINMAXDIST is smaller than the distance to the farthest current result will be visited.

Methods derived from the R-tree

We now devote the rest of the section to indexing structures that, although derived from the R-tree, significantly depart from its basic paradigm, by using partitioning elements other than hyperrectangles, by storing pointers to database elements in internal nodes, or by allowing multiple pointers to the same database elements. These methods are often tailored to specific types of queries.

Some data types are not well-suited for indexing with R-trees or multidimensional indexing structures based on recursive decomposition. These are typically objects whose bounding box is significantly more elongated in one dimension than in another. An example are segments aligned with the coordinate axes. A paradigm was introduced by Kolovson and Stonebraker [103] aimed at solving this difficulty. The resulting class of indexing approaches, called *Segment Indexes*, can be used to modify other existing techniques. When the method-

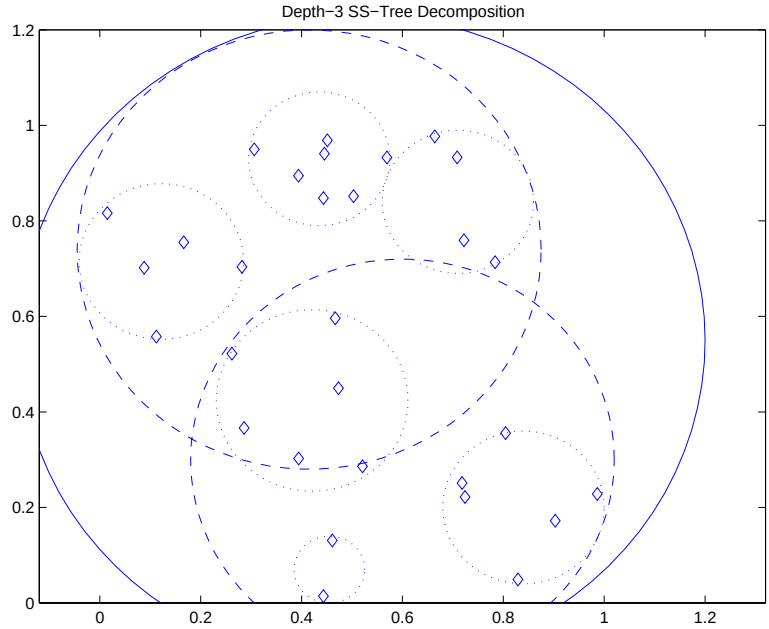

Figure 8: Two-dimensional space decomposition using a depth-3 SS-tree. Database vectors are denoted by diamonds. Different line types correspond to different levels of the tree. Starting from the root, these line types are: solid, dash-dot, dashed and dotted. The dataset is identical to that of Figure 3.

ology is applied to an R-tree, the resulting structure is called a *Segment R-tree* or *SR-tree*. SR-trees can store data items at different levels, not only at leaves. If a segment spans the rectangles of several nodes, it is stored in the one closest to the root. If necessary, the end portions of the segment are stored in other nodes. A variation of the index is the *Skeleton SR-tree*, where the data domain is recursively partitioned into non-overlapping rectangles using the distribution of the entire database, and items are subsequently inserted in any order. The resulting tessellation of the search space resembles more closely that of a non-balanced quadtree than that of an R-tree. Experimental results in 2 dimensions show that the SR-tree has essentially the same performance as the R-tree, while 10-fold gains can be obtained using the Skeleton version.

When the restriction that the hyperplanes defining a hyperrectangle be perpendicular to one of the cartesian coordinate axes is removed, a bounding polyhedron can be used as the base shape of an indexing scheme. Examples are the *Cell-tree* [84], and the P-tree [93]. In the *Polyhedron-tree* or *P-tree*, a set of vectors V, of size larger than the number of dimensions of the space, is fixed, and bounding hyperplanes are constrained to be orthogonal to these vectors. The constraint significantly simplifies the index construction, maintenance and search operations over the case where arbitrary bounding hyperplanes are allowed, while yielding tighter bounding polyhedra than the traditional R-tree. Two properties are the foundations of the approach. First, one can map any convex polyhedron having n faces into a $m-$dimensional hyperrectangle, where $m \leq n \leq 2m$. The corresponding $m-$dimensional

space is called orientation space. Thus, if the number of vectors in V is m, and the constraint on the hyperplanes is enforced, any bounding polyhedron can be mapped into a $m-$dimensional hyperrectangle, and indexing methods that efficiently deal with rectangles can then be applied. The second property is that intersection of bounding polyhedra in the feature space (again assuming that the constraint is satisfied) is equivalent to intersection of the corresponding hyperrectangles in orientation space. A P-tree is a data structure based on a set of orientation axes, defining a mapping to an orientation space, and an R-tree which indexes hyperrectangles in orientation space. Compared to a traditional R-tree, the P-tree trades off dimensionality of the query space, resulting in "curse of dimensionality" type of inefficiencies, for (potentially significantly) better coverage, resulting in better discrimination of the data structure. Algorithms exist to select the appropriate axes, and to manage the data structure. Variations of the scheme rely on different variants of the R-tree, and one could conceivably construct P^+-trees, P^*-trees, P^X-trees (using X-trees) etc. Due to the higher dimensionality of the orientation space, the P-tree is better suited to representing two or three-dimensional objects (for instance in GIS applications) than high-dimensional features.

When used for nearest-neighbor queries, R-tree-like structures suffer from the fact that hyperrectangles are poor approximations of high-dimensional Minkowsky balls (except for $D^{(\infty)}$), and, in particular, of Euclidean balls. The *similarity search tree* or *SS-tree* [165] is a variation of the R*-tree that partitions the search space into hyperspheres rather than hyperrectangles. The nearest-neighbor search algorithm adopted is the one previously mentioned [145]. On $2-$ to $10-$dimensional synthetic datasets, containing 100,000 points (Normally and Uniformly distributed), the SS-tree has storage utilization of about 85%, compared to the $70\% - 75\%$ achieved by the R*-tree. The index construction also requires 10% to 20% CPU time. During searches, when $d > 5$, the SS-tree is almost invariably faster than the R*-tree, though it appears to suffer from a similar performance degradation as the dimensionality of the search space increases.

While hyperrectangles are not particularly well-suited to represent Minkowsky balls, hyperspheres do not pack particularly well, and cover too much volume. These considerations were used by Katayama and Satoh [98] to develop the *Sphere-Rectangle Tree*, also called *SR-tree* (not to be confused with the SR-tree in [103]). The partitioning elements are the intersection of a bounding rectangle with a bounding sphere. Pointers to the data items are contained in the leaves, while intermediate nodes contain children descriptors, each consisting of bounding rectangle, a bounding sphere, and the number of points indexed by the subtree rooted at the child. The insertion algorithm is similar to that of the SS-tree, except that here both bounding sphere and bounding rectangle must be updated. The center of the bounding sphere of a node is the centroid of all the database elements indexed by the subtree rooted at the node. To determine the radius of the bounding spheres, two values are computed: the first is the radius of the sphere containing all the bounding spheres of the children nodes, the second is the radius of the sphere containing the bounding rectangles of the children. The radius of the bounding sphere is defined as the minimum of these values. The space refinement approach is similar in spirit to that of the P-tree [93]. The tree construction

is less CPU intensive than that of the R*-tree and of the SS-tree, however the number of required disk accesses is larger. During nearest-neighbor search, the methods outperforms the SS-tree as it requires up to 33% less CPU time, 32% fewer disk accesses, and visits a smaller number of leaves. Only smaller gains are observed over the VAMSplit R-tree, and only for certain datasets, while for other datasets, the SR-tree can be significantly worse.

All the variants of the R-tree examined so far partition the space using all the dimensions simultaneously. The *Telescopic-Vector tree* or *TV-tree* [108] is a variant of the R*-tree that attempts to address the curse of dimensionality by indexing only a selected subset of dimensions at the root, and increasing the dimensionality of the indexed space at intermediate nodes whenever further discrimination is needed. Each internal node of the tree considers a specified number of active dimensions, α, and corresponds to an α-cylinder, that is, to an infinite region of the search space whose projection on the subspace of the active dimensions is a sphere. The number of active dimensions is not fixed, but changes with the distance from the root. In general, higher levels of the tree will use fewer dimensions, while lower levels use more dimensions to achieve a desired discrimination ability. The tree need not be balanced, and different nodes at the same level generally have different active dimensions. The strategy proposed in the original paper to select the active dimensions consists of sorting the coordinates of the search space in increasing order of discriminatory ability through a transformation (such as the Karhunen-Loève Transform, the Discrete Cosine Transform [139], or the wavelet transform [56]), and specifying the number α of active coordinates at each node. The α most discriminatory coordinates are then used. The search algorithm relies on the branch-and-bound algorithm of Fukunaga and Narendra [78]. Experiments on a database, in which words are represented as 27-dimensional histograms of letter counts, show that the scheme scales very well with the database size. In the experiments, the best value of α appears to be 2. The TV-tree accesses 1/3 to 1/4 of the leaves accessed by the R*-tree for exact match queries.

C Projection-Based Methods

The projection-based indexing methods (Table 4) support nearest-neighbor searches. We can divide them into at least two classes: indexing structures returning results that lie within a prespecified radius of the query point, and methods supporting approximate $(1+\epsilon)$ queries.

C.1 Approaches Supporting Fixed-Radius Nearest-Neighbor Searches

The first category is specifically tailored to fixed-radius searches, and was originally introduced by Friedman, Baskett and Shustek [74]. Friedman's algorithm projects the database points onto the individual coordinate axes, and produces d sorted lists, one per dimension. In response to a query, the algorithm retrieves from each list the points whose coordinate lie within r of the corresponding coordinate of the query point. The resulting d sets of candi-

Author and Reference	**k-nn Seach Time**	**Space Requirement**	**Construction Time**	**Query Type**
Friedman et al. [74]	ndr	nd	$dn \log n$	fixed radius (r)
Yunck [171]	ndr	-	-	fixed radius
Nene and Shree* [123, 124]	$nd + n\frac{1-r^d}{1-r}$	nd	$dn \log n$	fixed radius (r)
Agarwal† and Matoušek [2]	$n \log^3 n m^{-1/\lceil d/2 \rceil}$ $+k \log^2 n$	$m^{1+\delta}$	$m^{1+\delta}$	ray tracing: exact
Meiser [119]	$d^5 \log n$	$n^{d+\delta}$	-	exact
Kleinberg [100]	$(d \log^2 d)$ $\cdot(d + \log n)$	$n \cdot (d \log^2 d \cdot n^2)^d$ $+(d \cdot \log^2 d \cdot n^2)^{d+1}$	$n \cdot (d \log d)^2$	$1+\epsilon$, probabilistic
Kleinberg [100]$^{\triangle}$	$n + d \cdot \log^3 n$	$d^2 \cdot n \cdot p(n)$	$d^2 \cdot n \cdot p(n)$	probabilistic
Indyk and Motwani [92]	$d \cdot n^{1/\epsilon}$ $+\mathrm{Pl}(n)$	$n\ \mathrm{Pl}(n)$	$n^{1+1/\epsilon} + d \cdot n$ $+\mathrm{Pl}(n)$	$1+\epsilon$
Indyk and Motwani [92]	$d + \mathrm{Pl}(n)^{\oplus}$	$n \cdot \mathrm{Pl}(n)$	$[n + \mathrm{Pl}(n)]$ $\cdot(1/\epsilon)^d$	$1+\epsilon$, $(0<\epsilon<1)$

Table 4: Comparison of projection-based methods. All the values are order $O(\cdot)$ results.
$*$ The expression for Nene and Shree's search time assumes that the database points are uniformly distributed in the unit $d-$dimensional hypercube.
† The expressions for Agarwal and Matoušek results hold for each $m \in \left[n, n^{\lceil d/2 \rceil}\right]$; δ is an arbitrarily small but fixed value.
$\triangle$ In the second method by Kleinberg, $p(n)$ is a polynomial in n.
$\oplus$ The term "Pl(n)" denotes a polynomial in $\log(n)$.
$\circ$ The method works for D^p distances, $p \in [1, 2]$.

date points are finally searched exhaustively. If the data points are uniformly distributed, the complexity of the search is roughly $O(ndr)$. An improvement to the technique was proposed by Yunck [171], but the overall complexity of the search is still $O(ndr)$.

In the same spirit as Friedman [74], Nene and Shree [123, 124] order the database points according to their values along individual coordinates, and for each ordered list maintain a forward mapping from the database to the list and a backward mapping from the list to the database. In response to a nearest-neighbor query, the first list is searched for points whose first coordinate lies within r of the 1st coordinate of the query point. This candidate set is then pruned in $d-1$ steps. At the ith iteration, the ith forward map is used to remove those points whose ith coordinate does not lie within r of the ith coordinate of the query. The procedure returns a set of points that lie within a hypercube of side $2r$ centered at the query point, which are then searched exhaustively. The selection of the parameter r is critical: too small and the result set will be empty most of the times, too large and the scheme does not perform significantly better than Friedman's. Two selection strategies are suggested. The first finds the smallest hypersphere that will contain at least one point with high (prespecified) probability, and uses the radius of the hypersphere as the value of r. As discussed in Section 2.4 the drawback of the approach is that the hypercube returned by the search procedure quickly becomes too large as the number of dimensions increases. The second approach directly finds the smallest hypercube that contains at least a point with high, prespecified, probability. Experiments on a synthetic dataset containing $30,000$ points show that the proposed algorithm is comparable to or slightly better than the k-d-tree for $d < 10$, and significantly better in higher dimensionality spaces. For $d = 15$, where the k-d-tree has the same efficiency as linear scan, the proposed method is roughly 8 times faster. Interestingly, for $d \in [5, 25]$ the difference in search time between exhaustive search and the proposed method does not vary significantly. As the database size grows to $100,000$, the k-d-tree appears to perform better for $d \leq 12$ when the data is normally distributed and for $d \leq 17$ for normally distributed data. Beyond these points, the k-d-tree performance quickly deteriorates and the method becomes slower than exhaustive search. Nene's method, however, remains better than exhaustive search over the entire range of analyzed dimensions. The approach appears to be better suited for pattern recognition and classification than for similarity search in feature space, where often no results would be returned.

A substantially different projection method was proposed by Agarwal and Matoušek [2]. Their algorithm maps $d-$dimensional database points $\mathbf{x}$ into $d+1-$dimensional hyperplanes through the functions

$$h_{\mathbf{x}}(\mathbf{t}) = 2\mathbf{t}[1]\mathbf{x}[1] + \ldots + 2\mathbf{t}[d]\mathbf{x}[d] - (\mathbf{x}[1]^2 + \ldots + \mathbf{x}[d]^2).$$

The approach relies on a result from [60], which says that $\mathbf{x}$ is closest point in the database $\boldsymbol{\mathcal{X}}$ to the query $\mathbf{q}$ if and only if

$$h_{\mathbf{x}}(\mathbf{q}) = \max_{\mathbf{y} \in \boldsymbol{\mathcal{X}}} h_{\mathbf{y}}(\mathbf{q}).$$

Then, nearest-neighbor searching is equivalent to finding the first hyperplane in the upper envelope of the database hit by the line parallel to the $(d+1)$st coordinate axis, and passing

through the augmented point $[\mathbf{q}[1], \ldots, \mathbf{q}[d], \infty]$. The actual algorithms rely on results from [117]. Another algorithm based on hyperplanes is described by Meiser [119].

C.2 Approaches Supporting $(1+\epsilon)$ Nearest-Neighbor Searches

Data structures based on projecting the database onto random lines passing through the origin are extremely interesting members of this class. A random line is defined as follows: Let d be the dimensionality of the search space, let $\mathcal{B}_0(1)$ be the unit radius ball centered on the origin, and call $\mathcal{S}$ be its surface. Endow $\mathcal{S}$ with the uniform distribution, and sample a point accordingly. Then a random line is defined as the line that passes through the random point and the origin of the space. The fundamental property on which methods of this class rely is that, *if a point* $\mathbf{x}$ *is closer in Euclidean distance to* $\mathbf{y}$ *than to* $\mathbf{z}$*, then, with probability greater than* $1/2$*, its projection* $x^{'}$ *on a random line is closer to the projection* $y^{'}$ *than to the projection* $z^{'}$. Given ϵ and an acceptable probability of error, the index is built by independently sampling L vectors $\{v_\ell\}$ from the uniform distribution on the sphere $\mathcal{S}$. L is chosen to be of order $d \log^2 d$, where the multiplicative constants depend on ϵ and the probability of error. If $\mathbf{x}_1, \ldots, \mathbf{x}_N$ are the points in the database, the collection of midpoints $\{\mathbf{x}_{ij} = (\mathbf{x}_i + \mathbf{x}_j)/2 \mid 1 \leq i, j \leq N\}$ is then generated. For each random vector v_l, the $\mathbf{x}_{ij}$ are ordered by the value of their inner product with v_ℓ, thus producing L lists $S_i, \ldots, S_\ell$. A pair of entries in a list is called an interval, a pair of adjacent entries is a *primitive interval*, and a sequence of L primitive intervals, one from each list, is called a trace. A trace is realizable if there exist a point in $\mathbb{R}^d$ such that its inner product with the ℓth vector v_ℓ lies in the ℓth primitive interval of the trace, for all values of ℓ. For each realizable trace, build a complete directed graph on the database with an edge from $\mathbf{x}_i$ to $\mathbf{x}_j$ if $\mathbf{x}_i$ dominates $\mathbf{x}_j$ in more than $L/2$ lists, and from $\mathbf{x}_j$ to $\mathbf{x}_i$ otherwise. $\mathbf{x}_i$ dominates $\mathbf{x}_j$ in list S_ℓ with respect to a trace, if, in S_ℓ, p_{ij} lies between the ℓth primitive interval of the trace and the point p_{jj} (equal to p_j). The nodes of the directed graph are ordered in such a way that there is a path of length at most 2 from each node to any of the ones following it in the ordering (called an apex ordering). Each realizable trace σ is stored together with the corresponding apex ordering S^*_σ. Note that there are $O(n \log d)^{2d}$ realizable traces. Index construction requires the computation of nL inner products in d dimensions, enumerating the realizable traces (which requires $O(L^d n^{2d})$ operations [60]) computing the corresponding digraph ($O(Ln^2)$) and storing the derived sorted list ($O(n)$). Note the exponential dependencies on the number of dimensions. To search for neighbors of the query point q, for each ℓ the inner product $v_\ell \cdot q$ is computed and its position in the list S_ℓ found by binary search. The primitive interval σ^q_ℓ of S_ℓ in which $v_\ell \cdot q$ falls is determined, and the trace $\sigma^q_1 \ldots \sigma^q_L$ constructed. The corresponding apex ordering S^*_σ is then retrieved and its first k elements are returned. Thus, a query involves computing L ($O(d \log^2 d)$) inner products in d dimensions, performing L binary searches on n items, a lookup in a large table, and reading the first k entries of a list; thus, the query time is $O((d \log d^2)(d + \log n))$. The query processing is deterministic. The index construction, however is based on the selection of L random vectors. One can show that, with probability that can be made arbitrarily small, the set of vectors can lead to an

erroneous index. If the set of vectors is correct, all the resulting retrievals will be exact.

Kleinberg [100] proposes a probabilistic method to find $(1+\epsilon)$ nearest-neighbors using random projections.

Kleinberg [100] also proposes a randomized algorithm that removes the exponential dependence of the preprocessing in the number of dimensions, and that relies a randomization step in the query processing. The method is however less efficient while processing a query.

Indyk and Motwani [92] introduce the *Ring-Cover Trees*. Nearest-neighbor queries are reduced to point location in equal balls: given a database $\boldsymbol{\mathcal{X}} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, fix a radius r, consider the balls $\mathcal{B}_{\mathbf{x}_1}(r), \ldots, \mathcal{B}_{\mathbf{x}_n}(r)$, and construct a data structure that returns $\mathbf{x}_i$, if a query point q belongs to any $\mathcal{B}_{\mathbf{x}_i}(r)$, and returns a FAILURE otherwise. The $\epsilon-$approximation of the problem consists of returning $\mathbf{x}_i^{'}$ rather than $\mathbf{x}_i$ if $q \in \mathcal{B}_{\mathbf{x}_i}(r)$, where $\mathbf{x}_i^{'}$ is such that $q \in \mathcal{B}_{\mathbf{x}_i^{'}}(r+\epsilon)$. To describe the Ring-Cover Tree, the following definitions are needed. A *ring* $R(\mathbf{x}, r1, r2)$ where $r1 < r2$ is the difference $\mathcal{B}_{\mathbf{x}}(r2) - \mathcal{B}_{\mathbf{x}}(r1)$. A ring $R(\mathbf{x}, r1, r2)$ is $(\alpha_1, \alpha_2, \beta)-$*ring separator* for the database $\boldsymbol{\mathcal{X}}$ if $|\boldsymbol{\mathcal{X}} \bigcap \mathcal{B}_{\mathbf{x}}(r_1)| \geq \alpha_1 |\boldsymbol{\mathcal{X}}|$, and $|\boldsymbol{\mathcal{X}} \setminus \mathcal{B}_{\mathbf{x}}(r_2)| \geq \alpha_2 |\boldsymbol{\mathcal{X}}|$. A set $S \subset \boldsymbol{\mathcal{X}}$ is a $(\gamma, \delta)-$*cluster* if for each $\mathbf{x} \in S$, $\left|\boldsymbol{\mathcal{X}} \bigcap \mathcal{B}_{\gamma\rho(S)}(\mathbf{x})\right| \leq \delta |\boldsymbol{\mathcal{X}}|$, where $\rho(S)$ is the maximum distance between the farthest points in S. A sequence $S_1, \ldots, S_l$ of subsets of $\boldsymbol{\mathcal{X}}$ is a *(b,c,d)-cover* for $S \subset \boldsymbol{\mathcal{X}}$ of there exists $r > d \cdot \rho\left(\bigcup_i S_i\right)$, such that $S \subset \bigcup_i S_i$ and $\left|\boldsymbol{\mathcal{X}} \bigcap \left(\bigcup_{\mathbf{x} \in S_i} \mathcal{B}_r(\mathbf{x})\right)\right| \leq b \cdot |S_i|$ and $|S_i| \leq c \cdot |\boldsymbol{\mathcal{X}}|$. The Ring Cover tree is constructed by recursively decomposing the database $\boldsymbol{\mathcal{X}}$ (which corresponds to the root) into smaller sets $S_1, \ldots, S_l$, which become nodes of the tree. There are two cases: either a non-leaf node has an $(\alpha, \alpha, \beta)-$ ring separator, in which case the node is a ring node and is split into its intersection and its difference with the outer ball of the ring; or the node has a $(b, \alpha, d)-$cover, in which case it is called a cover node and is split into $l+1$ nodes, the first l of which correspond to its intersections with $\bigcup_{\mathbf{x} \in S_i} \mathcal{B}_r(\mathbf{x})$,$(1, \ldots, l,)$ and the last contains all the remaining items (residual node). Ring nodes and cover nodes also contain different information on how their descendants are obtained. Searching a ring node corresponds to checking whether the query is inside the outer sphere of the separator, and searching the appropriate child. Searching a cover node is performed by identifying the set S_i to which the query belongs (i.e., the query is within a sphere of appropriate radius centered at one of the elements of S_i). If such set is identified, then the corresponding subtree is searched. If no such S_i exists, but the query is close to S_j, then both S_j and the residual set are searched, and the closest of the two results is returned. Otherwise, the residual set is searched. Two algorithms are proposed to index cover nodes; the first is based on bucketing, and is similar to the Elias algorithm described in [163]; the second is based on *locality-sensitive hashing*. The search algorithm for the Ring Cover tree is polynomial in $\log(n)$ and in d.

Miscellaneous Partitioning Methods In this section, we describe three recent partitioning methods that attempt to reduce the curse-of-dimensionality in different ways: CSVD, the Onion index and the pyramid of Berchtold, Böhm, and Kriegel. All these methods are specifically designed to support a particular class of queries.

A Clustering with Singular Value Decomposition: CSVD

Most dimensionality reduction approaches are either computationally intensive (multidimensional scaling), or rely on the properties of the entire database (SVD followed by variable-subset selection). To efficiently capture the local structure of a database without incurring the high costs of multidimensional scaling, *CSVD* [157, 156] first partitions the data into homogeneous groups, or clusters, and then separately reduces the dimensionality of each group. Clustering is accomplished either with optimal methods, such as LBG [109], or with faster suboptimal schemes, such as tree-structured vector quantizers [140]. Dimensionality reduction of each individual cluster is performed using SVD followed by variable-subset selection. Data points are then represented by their projection on the subspace associated with the cluster they belong to. The scheme can be recursively applied.

The index is represented as a tree. Each node contains the centroid of the cluster, its radius (defined as the distance between the centroid and the farthest point of the cluster), and the dimensionality reduction information, namely, the projection matrix and the number of retained dimensions. Non-leaf nodes contain the partitioning information, used to assign a query vector to its corresponding cluster, and pointers to the children, each of which represents a separate cluster. Terminal nodes contain an indexing scheme supporting nearest-neighbor queries, such as the ordered partition [99].

Nearest-neighbor queries are executed by descending the tree and identifying the terminal cluster to which the query point belongs, called the primary cluster. The query point is projected onto the subspace of the primary cluster, the within-cluster index is searched, and an initial set of results retrieved. The Pythagorean theorem is used to account for the distance between the query point and the subspace of the primary cluster. Then, a branch-and-bound algorithm is applied to identify other candidate nodes. The radiuses are used to discard clusters that cannot contain any of the $k-$nearest-neighbors, and the other clusters are visited in order of the distance between centroids and query point.

The method yields exact point queries and approximate $k-$nearest-neighbor queries. The number of cluster is selected empirically. Different algorithms guide the dimensionality reduction. The user can either specify the desired average number of dimensions to retain, a desired normalized mean squared error of the approximation, or a desired precision/recall target.

Experiments on a database containing $160,000$, 60-dimensional texture feature vectors show that, when performing $20-$nearest-neighbor queries, CSVD using 32 clusters is 100 times faster than sequential scan if the desired recall is .985, and 140 times faster if the desired recall is .97.

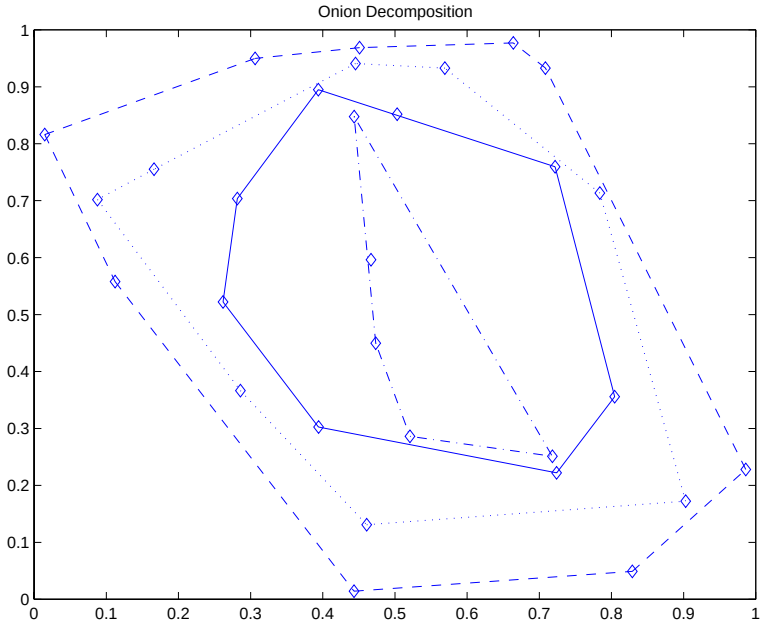

Figure 9: Two-dimensional space indexing using the onion data structure. Database vectors are denoted by diamonds. The dataset is identical to that of Figure 3.

B The Onions Index

While most queries belong to the three main categories described in Section 2.2, sometimes the user wants to retrieve, from the databases, records that maximize a particular scoring function.

The Onion technique [45] retrieves from a database the points maximizing a linear or convex scoring function. The main property on which the method relies is that points maximizing a convex function belong to the convex hull of the database. The index is then constructed by iteratively computing the convex hull of the dataset, and removing all its points. The database is therefore "peeled" in layers like an onion, hence the name. Points belonging to each layer are stored in a separate list. Figure 9 contains an example.

A query is specified by providing a scoring function and the number of desired results. If the function is linear, only the coefficients associated with each dimension are required. The search starts by sequentially searching the outermost layer, and proceeds towards the center, while maintaining a list of current results. The search terminates when the list of current results is not modified while searching a layer. Index maintenance algorithms exist, but the index usually must be recomputed after a large number of insertions or deletions.

C The Pyramid Technique

All the described recursive decomposition methods are affected by the curse of dimensionality. Berchtold, Böhm, and Kriegel [18] propose a method, the *pyramid technique*, which partitions the space into a number of regions that grows linearly in the number of dimensions. The d-dimensional search space is partitioned into $2d$ pyramids having vertices at the origin and height aligned with one of the coordinate axis. Each pyramid is then cut into slices parallel to the base, which correspond to a page in the index. The slices of each pyramid can then be indexed using a B^+-tree.

Insertions are very simple, and so are point queries, performed by identifying the appropriate pyramid, using the B^+-tree to find the right slice, and exhaustively searching the corresponding page. Range queries, however, are computationally complex, since they entail identifying which pyramids intersect the query rectangle.

Experimental results, carried out on 1 million uniformly distributed synthetic data in up to 100 dimensions, compare the pyramid technique to the X-tree. For cubic query regions, the pyramid is about 2500 times faster than the X-tree in 100 dimensions, (where the X-tree is probably significantly slower than linear scan.) When the query box is a hyperrectangle restricted in $n^{'}$ dimensions, and fully extended in $d - n^{'}$ dimensions, the pyramid is still faster than linear scan. For example, for $d = 100$ and $n^{'} = 13$, the pyramid is about 10 times faster than sequential scan.

Metric Space Methods This section describes metric space indexes, the 2nd main class of indexing methods. The defining characteristics is that the space itself is indexed, rather than the individual database elements.

A The Cell Method

Berchtold, Erlt, Keim, Kriegel and Seidl [19] precalculate an index for the 1-nearest-neighbor *search space* rather than for the points in the space. Given a database $\boldsymbol{\mathcal{X}}$, solving the nearest-neighbor problem is equivalent to computing a tessellation of the search space into Voronoi regions [10], and identifying the region to which a query vector belongs. In general, Voronoi regions are complex polyhedra, which are difficult to represent in a compact form and to store in a database. The solution proposed in [19] consists of approximating the Voronoi regions by their minimum bounding hyperrectangle (MBH) and storing the MBHs in a X-tree. Since the computation of the exact MBH can be quite time consuming, a slightly suboptimal linear programming method is suggested: to construct the Voronoi region of a particular point, the method use only those database elements that lie within a prespecified distance, those whose MBHs intersect, and the nearest neighbors in each of the coordinate directions.

Since MBHs of neighboring points overlap, a decomposition technique is proposed that

yields a better approximation of the actual Voronoi regions, thus reducing the overlap. Each MBH is successively "carved out" in a different dimension.

Experimental results on 10000 4- to 16-dimensional synthetically generated uniform data points show that this technique is between 1.01 and 3.5 times faster than the R*-tree and up to 2 times faster than X-tree. On 8-dimensional real data, the method is four times faster than the X-tree alone.

B Vantage Point Methods

Vantage point methods are a relatively recent class of approaches, though their origins could be found as early as in [34]. Yianilos introduced the *Vantage Point Tree* or *vp-tree* in 1986-87, apparently as a development of the work in [169]. An analogous data structure was developed by Uhlmann and called *Metric Tree* [161, 160]. A vp-tree [170] relies on pseudometrics. First, the distance function is remapped to the range $[0, 1]$, by either scaling (if the distance is bounded) or through the well known formula $\overline{D}(\mathbf{x}, \mathbf{y}) = D(\mathbf{x}, \mathbf{y}) / [1 + D(\mathbf{x}, \mathbf{y})]$. Then a point $\mathbf{v}$ (*vantage point*) is selected. To construct the index, the database points are then sorted according to their scaled distance from $\mathbf{v}$ (i.e., in ascending order of $\overline{D}(\cdot, \mathbf{v})$), the median scaled distance is computed, and the points having scaled distance smaller than the median are assigned to the *left subspace* of $\mathbf{v}$, while the remaining ones are assigned to the *right subspace*. The procedure is recursively applied to the left and right subspace. Different selection and termination criteria define different versions of the indexing structures. The simplest vp-tree relies on simple operations to select the appropriate vantage point among a random subset of the points associated with each node of the tree. Enhanced versions use a pseudometric, defined as $D_{\mathbf{v}}(\mathbf{x}, \mathbf{y}) = \left|\overline{D}(\mathbf{x}, \mathbf{v}) - \overline{D}(\mathbf{y}, \mathbf{v})\right|$, the key property of which is that $D_{\mathbf{v}}(\mathbf{x}, \mathbf{y}) \leq \overline{D}(\mathbf{x}, \mathbf{y})$, to map the original space into a lower-dimensional Euclidean space, each coordinate of which corresponds to a different node in the path between the root and a leaf.

The algorithm returns fixed-radius nearest-neighbors, where the radius ρ is determined at query time. The search is performed using a branch-and-bound method, which discards subtrees whose distance to the query point is larger than that of the current result or than the radius. Theoretical arguments suggest that search is possible in $O(d \log n)$ time. Experiments show that on synthetic data with limited structure, vp-trees have very similar performances to k-d-trees in up to 15 dimension. When the space has a structure, and in particular when the intrinsic dimensionality is much smaller than the actual dimensionality, the vp-trees are 1 to 2 orders of magniture faster than the k-d-trees for $d > 10$, and the gap appear to increase with the dimensionality. When applied to image snippet retrieval, on a database of about 1 million images of size 32×32, the vp-tree appears to visit on average only 5% of the nodes.

C M(S,Q) and D(S)

Clarkson [52] considers the following problem: given a set $\boldsymbol{\mathcal{X}}$ (the universe), an appropriate distance function on pairs of elements of $\boldsymbol{\mathcal{X}}$, and a database $S \subset \boldsymbol{\mathcal{X}}$, build a data structure on S that returns efficiently the closest element of S to any query point $q \in \boldsymbol{\mathcal{X}}$.

$M(S, Q)$ is a fast, approximate data structure, which supports probabilistic queries. $M(S, Q)$ is constructed using a set Q of representative query points, which can be provided a-priori, or constructed at query time using the user input. For each point (or site) p_j in S, $M(S, Q)$ maintains a list of other points $\mathcal{D}_j$. $\mathcal{D}_j$ is built as follows: consider a subset R of the database, construct a random ordering of its points, and let R_i be the collection of the first i points according to this ordering. Consider a sequence Q_i of randomly selected subsets of Q, having size $(K \cdot i)$. If p_k is the nearest element of R_{i-1} to $q \in Q_i$, and p_j is a $(1+\epsilon)$- neighbor of q, then p_j is added to $\mathcal{D}_i$. Searching $M(S, Q)$ consists of starting at point p_1, walking $\mathcal{D}_1$ until a site closer to the query than p_1 is found, repeating recursively the same operation on the list of the newly found point, until a list $\mathcal{D}^*$ is found which does not contain closer elements to the query than its corresponding point p^*. By fine tuning the parameters of the algorithm, and selecting ϵ appropriately, one can show that probability of failure to return the nearest neighbor of the query point is $O(\log n^2/K)$.

A different structure, called *D(S)* is also proposed. The index construction is recursive, and proceeds as follows. Select a random subset R of S. Associate with each element y of R an initially empty list, L_y, and two initially empty sets, S_y^1 and S_y^3. The list L_y contains the other elements of R sorted in non-decreasing distance from y. For each elements x of $S \setminus R$ (i.e., belonging to S but not to R), find its nearest neighbor in R, say y, and add x to the set S_y^1. For each element x of S_y^1 compute its 3-nearest neighbors among the points of R. Add x to the sets S_y^3 belonging to its three nearest neighbors. Recursively construct $D(R)$ and $D(S_y^3)$ for each y. Recursion terminates either when a predefined depth is reached or when the number of elements is below a threshold. Leaves consists of a list of sites.

To find the nearest neighbor of a query point $\mathbf{q}$, recursively search $D(R)$ to find its nearest neighbor x within R. Walk the list L_x and retrieve the three closest points to $\mathbf{q}$ in R. Recursively search the structures $D(S_y^3)$ of the found neighbors, and return the nearest point to the query among the sites found by each search. When a leaf node is reached, perform sequential scan of the list. If $\Upsilon(S)$ is the ratio of the largest distance between distinct points of S to the smallest distance between distinct points of S, then the preprocessing time is $O\left(n(\log n)^{O(\log\log\Upsilon(S))}\right)$ and the query time is $(\log n)^{O(\log\log\Upsilon(S))}$.

D The M-tree

$\alpha-$cut queries can also be executed in metric spaces, and can be supported by indexing structures, of which the *M-tree* [49, 48, 172, 47] is an example. Internal nodes of an M-tree contain a collection of routing objects, while the database objects are stored in groups at the

leaves. The description of a routing object consists of the object itself, of its covering radius, defined as the maximum distance between the routing object and the objects stored at the leaves of its subtree (covering tree), the distance between the covering object and the covering object of its parent node, and a pointer to the root. The description of a database object stored within a leaf consists of the object itself, an identifier and the distance between the object and the covering object of its parent. The tree is constructed sequentially, by adding new items to the most suitable leaf, which is either the unique leaf covering the object or, in case of overlap, the leaf whose parent covering object is closest to the item. When leaves (or internal nodes) are full, they are split, the original covering object is discarded, and a new covering object is created for each of the new leaves. The basic algorithm consists of selecting new covering objects so that the corresponding regions have minimum overlap and minimum covering volume. Numerous routing object selection policies are proposed by the authors, including random selection, which incidentally is not significantly worse than other policies.

An $\alpha-$cut query that returns all the database entries at distance less than r from the query q starts at the root node, and recursively traverses the tree. At each node, the covering objects whose covering tree might intersect the search region, are identified and recursively searched, while the remaining covering objects are pruned. When a leaf is reached, it is scanned sequentially. The pruning algorithm uses $D(q, R_p)$, the distance between the query and the routing node of the parent, $D(R_n, R_p)$, the distance between the routing node being analyzed and the routing node of its parent, r, and r_n, the covering radius of the current node. A subtree is selected is selected if $|D(q, R_p) - D(R_n, R_p)| \leq r + r_n$.

The average number of distance computations and of I/O operations during a search appears to grow linearly in the dimensionality of the search space.

Compared to a R*-tree in terms of I/O cost for building the index and for performing square range queries covering 1% of the indexed space, the M-tree yields gains that increase approximately linearly with the number of dimensions. In terms of distance selectivity, the M-tree appears to be better than the R*-tree by a constant factor, in databases with up to 50 dimensions.

The M-Tree can also be used for k-nearest-neighbor queries, where it also outperforms the R*-tree.